A cost-aware logical framework

Yue Niu

November 2, 2023

CONTENTS

CHAPTER 1

INTRODUCTION

(1∗1) Among formal codifications of mathematics, dependent type theories stand out because they furnish simultaneously the substrate of constructions and the logical language governing those constructions. In type theory constructions are referred to as *terms* or *elements* while specifications are referred to as *types*. It is customary to write $a : A$ to mean that a is an element of the type A, *i.e.* a is a construction satisfying the specification A. In type theory, dependency is used to express the notion of families; in particular a *type family* is a family of types indexed in terms of a given type. An example of a type family is the family of vectors $\mathsf{vec}_A : \mathbb{N} \to \mathsf{Type}$ where each fiber $\mathsf{vec}_A(n)$ classifies sequences of type A of length $n : \mathbb{N}$.

(1∗2) In type theory the fundamental property of type equality is expressed by the rule of conversion: if A and B are equal types, then any element of A is an element of B, and vice versa. While type equality is often purely syntactic in non-dependent type theories, the situation is more complicated in dependent type theory because type equality is intertwined with the equality of *terms*. In the case of length-indexed sequences vec_A, the types $\mathsf{vec}_A(n)$ and $\mathsf{vec}_A(m)$ classify the same sequences just in case that n and m are equal as elements of $\mathbb{N}$. Because $\mathsf{vec}(n + m)$ and $\mathsf{vec}(m + n)$ should always classify the same elements, equality here refers to the *extensional* equality specification, *i.e.* n and m are equal when they denote the same number.

(1∗3) The mathematically natural notion of equality at higher types is the equality of *behavior* or input-output pairs. This is usually presented in type theory as the rule of function extensionality:

$$\textsc{FunExt}\quad \frac{\Gamma, x : A \vdash f(x) =_B g(x)}{\Gamma \vdash f =_{A\to B} g}$$

Consequently, constructions in type theory (such as type families) must respect the extensional/behavioral equality of functions. For instance, because insertion sort and merge sort are both sorting algorithms, they are equal as functions of type $\mathsf{list} \to \mathsf{list}$ and thus *indistinguishable* in type theory.

1.1. DEPENDENT TYPE THEORY FOR COST ANALYSIS

(1.1∗1) Applying the type-theoretic lens to computer science, constructions are rendered as programs and the logical language becomes the language of program specifications. As discussed in **(1∗3)**, one naturally speaks of *extensional/behavioral* equalities in dependent type theory. Thus, in the context of programs one obtains an equational theory for studying program *behavior*. However, programs differs from ordinary mathematical structures because it is common to speak of *different* programs that are extensionally equal.

(1.1∗2) The sense in which "equal" programs can be different may be traced back to Frege's analysis of *sense* and *reference*. Put crudely, Frege observes that although both the "morning star" and the "evening star" *refer* to the same celestial body, they are different modes of presentation and thus differ in their *senses*. Frege's observation highlights a similar phenomenon in the context of programs: as we mentioned in **(1∗3)**, although both insertion sort and merge sort are sorting algorithms (and thus extensionally equal), they evince different presentations that induce the same behavior, *i.e.* they possess different *intensional* structures.

(1.1∗3) An example of an intensional property is the *cost* of a program, which is an aspect of programs usually not amenable to direct investigation in dependent type theories. The obstruction stems from two conflicting constraints: on the one hand, one would like to accommodate as many extensionality principles as possible for ordinary mathematical reasoning; on the other hand, the addition of such principles (*e.g.* beta equivalence) undermines the cost structure of programs. To see this, consider a hypothetical function $\mathsf{cost} : \mathsf{bool} \to \mathbb{C}$ that assigns to each input computation its induced cost. Observe that there can only be two different possible costs $c_{\mathsf{tt}}, c_{\mathsf{ff}} \in \mathbb{C}$ in the image of cost corresponding to $c_{\mathsf{tt}} := \mathsf{cost}(\mathsf{tt})$ and $c_{\mathsf{ff}} := \mathsf{cost}(\mathsf{ff})$. But we undoubtedly want the ability to observe two distinct costs for two boolean computations that both result in tt!

(1.1∗4) In response to the problem in **(1.1∗3)** we may dispense with equivalences induced by computation. Consequently cost is not restricted to two possible outcomes because we do not necessarily have *e.g.* $(\lambda x.\, \mathsf{tt}) \star = \mathsf{tt}$. However, type dependency greatly complicates the usability of the resulting theory because the equality of types is intertwined with equality of computations. In practice one wants as many equalities as possible to be discharged automatically as a consequence of computation; for instance it would be impractical if the equality of the propositions $\mathsf{isEven}(1 + 1)$ and $\mathsf{isEven}(2)$ required a manual proof that needs to be transported at the use sites.

(1.1∗5) One possibility out of this dilemma is to eschew dependency, thereby trivializing the equational theory of types. Indeed, many current type-theoretic approaches [HDW17; Çiç+17; WWC17] to cost analysis employ simple type systems or refinement type systems. However, the paucity of the specifications available in these frameworks precludes analyses of algorithms that depend on sophisticated behavioral invariants of the data structures involved. For instance, to prove the quadratic cost bound for insertion sort, one has to know that the recursive call preserves length — an *extensional* property of endofunctions on lists that follows from the fact that insertion sort is a sorting algorithm. In general, we see that cost analysis depends on the behavioral/extensional properties of the programs and data structures involved; thus the lack of an equational theory means that simple type systems cannot provide a foundational account of cost analysis.

1.1.1. Operational notions of cost.

(1.1.1∗1) What is the cost of a program? One way to study the dynamic behavior of programs is by means of an *operational semantics*; thus, it is natural to define cost based on the operational behavior of programs. Following Plotkin [Plo81], a structural operational semantics (SOS) is a transition system $(S, \mapsto)$ where S is the set of possible states (in our context program states) and $\mapsto \subseteq S \times S$ is a relation on states. It is usually customary to distinguish a set of final/terminated states $\Omega \subseteq S$, so that $s \in \Omega$ denotes a terminal program. We define a transition step to be a pair (s, s') such that $s \mapsto s'$.

For deterministic transition systems (meaning that for any s there is at most one s' such that they form a transition step), one may define a partial function $\mathsf{cost} : S \rightharpoonup \mathbb{N}$ that assigns to a program s the number of transition steps it takes to reach a terminal state $s' \in \Omega$ or undefined if no such state exists.

(1.1.1∗2) The definition of cost proposed in **(1.1.1∗1)** is simple to work with, but a bit more flexibility is desirable for certain applications. For instance, one may only care about the occurrence of a subset of the transition steps (*e.g.* only function applications), a situation best encoded by starting from a *natural semantics* [Kah87] or *big-step* semantics. In essence, natural semantics sweeps away some implementation details of the SOS in the form of a *evaluation* relation $\Downarrow \subseteq S \times \Omega$ that directly relates a program to its terminal state. For many languages, a standard result [Har16] relates these two notions of operational semantics: $s \mapsto^* s'$ and $s' \in \Omega$ if and only if $s \Downarrow s'$, where $\mapsto^*$ is the reflexive-transitive closure of $\mapsto$.

(1.1.1∗3) One may endow natural semantics with the cost structure of programs in the style of Blelloch and Greiner [BG95], resulting in the notion of a *cost semantics* for a language. In a cost semantics the evaluation relation is upgraded to a ternary relation $\Downarrow \subseteq S \times \mathbb{C} \times \Omega$ such that $s \Downarrow^c s'$ means that s evaluates to s' using $c \in \mathbb{C}$ units of cost. Observe that a cost semantics may be parameterized over a choice for the cost structure $\mathbb{C}$. Because nearly all programming languages come equipped with the ability to compose computations, the cost structure needs to admit the structure of a semigroup $(\mathbb{C}, +)$ at the minimum. In practice, $\mathbb{C}$ is almost always required to form a monoid $(\mathbb{C}, +, 0)$ as well.

(1.1.1∗4) One can further increase the granularity of the cost structure of programs by introducing *cost annotations* [Hof19]. Here the idea is to allow the programmer to indicate where cost should be incurred in a program. Concretely this may be achieved via an operational semantics $\mapsto \subseteq (\mathbb{N} \times S) \times (\mathbb{N} \times S)$ that is instrumented with a cost counter that is incremented by $\mathsf{tick}(q)$:

$$(p, \mathsf{tick}(q)) \mapsto (\star, p + q)$$

Because the purpose of $\mathsf{tick}(q)$ is to record the incurrence of cost, its behavior is trivial.

(1.1.1∗5) While notions of program cost based on operational semantics is natural and admit (in)equational theories à la contextual equivalence, the resulting theory is not optimal for actually proving program equivalences or cost bounds. Indeed practitioners usually work with some form of upgraded *logical equivalence* that takes into account cost information [Çiç+17]. Although logical equivalence/logical relations are indispensable for studying the *metatheoretic* properties of a language, they do not provide a compositional theory for reasoning about cost, as stating or proving properties about open terms requires one to explicit quantify over closing instances. As we will discuss in Section 1.1.4, in a compositional theory of cost one views cost not as *properties* of raw programs but as *structures* attached to equivalence classes of program behavior.

1.1.2. Cost monads.

(1.1.2∗1) A compositional notion of cost well-adapted for type-theoretic manipulation can be gleaned from the practice of program profiling/instrumentation. More precisely, to profile/instrument is to embed an ordinary program in a *cost monad* $\mathbb{C} \times -$, where $\mathbb{C}$ encodes the cost structure as a monoid $(\mathbb{C}, +, 0)$.[1] An instrumented/cost-aware program is a pair of the form $(c, e) : \mathbb{C} \times A$ such that c is the cost assigned to the computation $e : A$. The monadic structure is given by the monoid structure of $\mathbb{C}$:

$$\begin{aligned} \mathsf{ret} &: \{A\}\ A \to \mathbb{C} \times A \\ \mathsf{ret}(a) &= (0, a) \end{aligned}$$

$$\begin{aligned} \mathsf{bind} &: \{A, B\}\ CA \to (A \to CB) \to CB \\ \mathsf{bind}((c, a), f) &= \mathsf{let}\ (c', b) = fa\ \mathsf{in}\ (c + c', b) \end{aligned}$$

(1.1.2∗2) The setup described in **(1.1.2∗1)** is a delicate arrangement because cost monads expose too much implementation detail: a cost-aware function $f_{\mathsf{inst}} : \mathbb{C} \times A \to \mathbb{C} \times B$ may simply ignore the cost component of its input. More importantly, one may define *exotic programs* — programs that have no coherent underlying behavior, and therefore cannot be said to be instrumented versions of an ordinary program. For instance, the following program chooses its behavior based on whether the input cost is zero:

$$\begin{aligned} f &: \mathbb{N} \times A \to \mathbb{N} \times \mathsf{bool} \\ f(0, a) &= (0, \mathsf{tt}) \\ f(\mathsf{suc}(n), a) &= (0, \mathsf{ff}) \end{aligned}$$

We say an encoding of cost-aware programs is *safe* when it is not possible to express such exotic programs. Put another way, the safety of an instrumentation of cost ensures that the *intension/cost* structure of programs do not interfere with their *extension/behavior*.

(1.1.2∗3) Although cost monads should not be used directly by users of a cost-aware programming language, they can be the target of a *cost-preserving translation* of a source language [Kav+19]. Safe encoding of cost structure follows from the fact that exotic programs lie outside the image of this translation.

[1]A cost monad induced by a monoid $\mathbb{C}$ is just the writer monad induced by $\mathbb{C}$.

(1.1.2∗4) We may hide the implementation of cost monads by working with an *abstract* monadic language for computational effects à la Moggi. The transparent definition of the cost monad $\mathbb{C} \times -$ is replaced by a unary type constructor C equipped with the monadic operations of $\mathsf{ret} : \{A\}\ A \to CA$ and $\mathsf{bind} : \{A, B\}\ CA \to (A \to CB) \to CB$. There are several ways of integrating the cost effect into this monadic language; for instance we may add a computation $\mathsf{inc} : \{A\}\ \mathbb{C} \to CA \to CA$ that increments the cost of a computation by a given cost. The semantics of the monadic language of cost effects is given by the Kleisli category associated with the cost monad. Consequently one obtains the expected commutation laws enjoyed by the transparent definition of cost monads; for instance in addition to the monad laws we have the following equation that expresses that inc commutes with bind:

$$\mathsf{bind}(\mathsf{inc}(c, e), f) = \mathsf{inc}(c, \mathsf{bind}(e, f))$$

The benefit of working in an abstract monadic language is that it is no longer possible to define exotic programs in the sense of **(1.1.2∗2)**: one may not observe the cost component of a computation $e : CA$.

1.1.3. The CBPV decomposition of cost structure.

(1.1.3∗1) Although monadic cost effects as described in **(1.1.2∗4)** are sufficient for reasoning about cost in a simply-typed setting, they do not integrate well into dependent type theory. One way to incorporate cost effects with type dependency is the **∂cbpv** calculus of Pédrot and Tabareau [PT19], a dependent version of Levy's call-by-push-value (CBPV) language. The primary feature of CBPV is that one maintains a dichotomy of values and computations at *both* the term and type level. The intuition is that computation types classify possibly effectful operations (such as incurring cost) while value types classify inert terms, a situation that Levy sums up as "a value is, a computation does".

(1.1.3∗2) The perspective of cost as an effect in CBPV was employed by Kavvos et al. [Kav+19] to formalize the method of recurrence extraction as a cost-preserving translation in the sense of **(1.1.2∗3)**. In this context cost is implemented by adding a single primitive effect $\mathsf{charge}_X(c, M)$ to the CBPV base language that may be read operationally as "charge c units of cost and continue as M". Notice the similarity between charge and inc as described in **(1.1.2∗4)**; the difference is that charge is defined at *every* computation type X. As we see in **(1.1.3∗3)**, inc_A may be recovered in the CBPV setting as $\mathsf{charge}_{\mathsf{F}(A)}$.

(1.1.3∗3) In the categorical semantics of CBPV, value types are interpreted as ordinary sets and computation types are interpreted as objects in the Eilenberg-Moore category associated to the cost monad $\mathbb{C} \times -$ induced by a given monoid $\mathbb{C}$; in other words a computation type X is interpreted as an algebra $(|X|, \alpha_X)$ over $\mathbb{C} \times -$. The value-computation dichotomy is bridged by a free-forgetful adjunction $\mathsf{F} \dashv \mathsf{U}$ in which the left adjoint sends a set A to the free $\mathbb{C} \times -$-algebra on A and the right adjoint forgets the algebra structure of a given algebra $(|X|, \alpha_X)$. The connection between the CBPV decomposition of the cost effect to the cost monad set up of **(1.1.2∗1)** can be seen by considering the associated Kleisli category, which may be defined as the full subcategory of the Eilenberg-Moore category spanned by the *free* $\mathbb{C} \times -$-algebras. Consequently, the equational theory of cost monads is inherited by the *free* computations in CBPV, *i.e.* computations of the form $\mathsf{F}(A)$.

1.1.4. Property *vs.* structure.

(1.1.4∗1) We see that operational notions of cost are not well-behaved in type theory because the tradition of operational semantics construes cost as a property of *raw terms*, a notion that ceases to exist *inside* type theory. On the other hand, the perspective of cost monad or cost effects renders cost as *structures* attached to *equivalence classes of typed terms*, a concept that is well-defined and enables a compositional theory of cost. The observation that notions ill-posed as properties can be objectified as structures is by now common in type theory research [SH21; AK16; SA21; Coq19]; for instance, one of the ideas employed in the normalization proof for cubical type theory [SA21] is to dispense with the notion of normal forms as properties of raw terms and instead view them as structures attached to judgmental equivalence classes of typed terms. In the context of cost analysis, we see that the move from properties to structures addresses the problem in **(1.1∗3)**: with both cost monads and cost effects it is not the case that the judgmental equivalence class of computations of type bool consists of only tt and ff.

1.1.5. Adequacy for cost.

(1.1.5∗1) *Uniform vs. non-uniform cost models.* Notions of cost derived from operational semantics as in Section 1.1.1 may be referred as *uniform/language-level* cost models in which the cost of a particular program construct is uniform across different algorithmic problems. In contrast, cost monads (Section 1.1.2) and cost effects (Section 1.1.3) naturally support *non-uniform/algorithm-specific* cost models in the sense that there is no inherent association of cost to program constructs;

rather the cost model is defined by the user of the framework and may vary across different problems.

It is helpful to identify the uniform and non-uniform models as meaningful in their own contexts. For the purposes of algorithm analysis it is clearly preferable to work inside a framework that allows for different cost models for different classes of problems; cost models in this sense cannot be detected at the level of operational semantics — how would one delineate a comparison or edge insertion operation? On the other hand, it is ill-formed to state theorems relating the cost semantics of different languages such as Kavvos et al. [Kav+19] with respect to a non-uniform cost model. However, we observe that by definition a uniform cost model is just an instance of a non-uniform model. Consequently, a language that supports a non-uniform cost model may be seen as the target of a cost-preserving translation of a language paired with a uniform cost model, a result known as an *adequacy theorem.*

(1.1.5∗2) Because both cost monads and cost effects possess compositional equational theories in the sense of **(1.1.1∗5)**, they constitute a form of *cost-aware* denotational semantics. Thus one may connect the uniform cost model induced by an operational semantics with the non-uniform cost model given by cost monads/cost effects via a variant of the classic adequacy theorem of Plotkin [Plo77]. Such a theorem is proven in the case of cost effects by Kavvos et al. [Kav+19]; more specifically, *op. cit.* prove a *bounding theorem* which states that one may extract from a PCF program e a semantic recurrence e_{rec} such that the operational cost of e is the same as the denotational/equational cost expressed by e_{rec}.

(1.1.5∗3) In Kavvos et al. [Kav+19] the bounding theorem is proven as a metatheorem relating two different languages. In a sufficiently expressive framework this metatheorem may be stated *internally.* When cost-preserving translations may be deduced as internal theorems, one can work with uniform cost models in a cost analysis framework that supports non-uniform cost models via the translation.

(1.1.5∗4) An adequacy result of the form described in **(1.1.5∗2)** may be appropriately characterized as a *relative* adequacy theorem because it is a statement about a cost semantics relative to another cost semantics (given by operational semantics). If the legitimacy of the *operational* notion of cost is in doubt, one may prove another adequacy theorem showing that the operational notion of cost may be realized via a *scheduler* on a given (abstract) machine model; a result of this form is generally referred to as a *Brent-type* theorem [Har12].

1.2. TOWARDS A COST-AWARE LOGICAL FRAMEWORK

(1.2∗1) What is missing from extant type theories dealing with cost analysis is the combination of composition, safety in the sense of **(1.1.2∗2)**, and the ability to express reasoning used in informal algorithm analysis **(1.1∗5)**. On the one hand, frameworks specifically designed for cost analysis such the various linear and refinement type systems [HAH12; WWC17; Raj+21] support composable cost bounds and are safe by design because cost structure is categorically isolated from programs. However, these systems do not consider program *equality*, a specification that is indispensable for the mathematical development of algorithm analysis.

On the other hand, one may work with libraries built on top of dependent type theories [HVH19; Dan08] that encode cost structure using some variation of the cost monad **(1.1.2∗1)**. By working inside dependent type theory one automatically has an equational theory, but as discussed in **(1.1.2∗2)** the cost monad does not provide a strong enough abstraction to rule out exotic programs. In light of recent work on integrating dependent types with computational effects [PT19], one may consider working with a dependent version of a CBPV language augmented with a cost effect as in **(1.1.3∗1)**. However, in both the monadic and CBPV setting it is unclear how one may naturally express *extensional equality* of programs.

(1.2∗2) The work of Kavvos et al. [Kav+19] addresses some limitations mentioned above by means of *stratification*: one translates the source program into a program in the language of *syntactic* recurrences, which is further interpreted into a semantic domain for mathematical manipulation. However, it is unclear how behavioral/extensional reasoning fits into the stratified framework.

(1.2∗3) *Extensional equality and the phase distinction.* One of the innovations of the present work is the observation that the safety aspect of the stratified framework of Kavvos et al. [Kav+19] may be expressed internally in type theory by the notion of a *phase distinction.* As I will discuss in Section 2.2.2, the (non)interaction between the intensional and extensional parts of a cost-aware program is completely analogous to the notion of the *phase distinction* in the theory of modules [HMM90]. While the original phase distinction was concerned with breaking dependence between dynamic and static components of program modules, the cost-aware setting evinces the phase distinction of *intension* and *extension*: the cost component of programs do not interfere with their behavior. In Section 2.2.3, I follow Sterling and Harper [SH21]'s synthetic reconstruction of the phase distinction and equip type theory with a program phase called the *extensional phase* in which cost structure is trivialized. The extensional phase may be internalized into the extensional modality $\bigcirc$ that extracts the extensional/behavioral content of a given type. Consequently, one may express the extensional equality of programs via the extensional equality type $\bigcirc(e_1 = e_2)$.

(1.2∗4) Given the state of affairs summarized above, I propose to develop a cost analysis framework satisfying the following criteria:

1. Composition: there is an equational theory of cost bounds.
2. Safety: it is not possible to express exotic programs **(1.1.2∗2)**.
3. Cost structure should not interfere with "normal mathematics"; in particular, one needs to be able to express the extensional equality of programs.

1.3. ANALYTIC AND SYNTHETIC THEORIES OF COST IN TYPE THEORY

(1.3∗1) In response to **(1.2∗4)** I outline two approaches to a type-theoretic treatment of cost analysis that we developed over the last several years [NH20; Niu+22]. In retrospect, the two approaches may be seen as an instance of the *analytic-synthetic* dichotomy.

(1.3∗2) *Analytic* vs. *synthetic theories.* Often times a mathematical structure may be profitably studied from two complementary perspectives. On the one hand, one may employ an *analytic* viewpoint that emphasizes the construction of the desired structure in terms of more primitive concepts. On the other hand, one may employ a *synthetic* viewpoint that axiomatically characterizes the desired structure. The prototypical theory evincing such a distinction is geometry: points and lines may be viewed analytically as constructions using cartesian coordinates or synthetically as primitive notions governed by certain axioms (*e.g.* Euclid's postulates).

(1.3∗3) A recent instance of the analytic-synthetic distinction in the context of type theory was exposed in Sterling and Harper's synthetic treatment of logical relations. Instead of defining logical relations from more primitive structures such as sets and families, *op. cit.* observes that the essence of the method of logical relations may be more directly appreciated by working in a theory in which *everything* is a logical relation. The structure of logical relations (*i.e.* their syntactic and semantic components) is governed by axioms akin to Euclid's postulates for geometry.

(1.3∗4) Traditionally, the analytic viewpoint underlies *computational type theories*, the most well-developed of which is the **Nuprl** family of type theories [All+06].

Here the primitive concept is *computation*, which is usually given as a programming language equipped with an operational semantics. Moreover, computation is the substrate from which all other type-theoretic concepts such as types/specifications, programs satisfying a specification, equality of programs and types are *constructed.* We will explore and develop this idea into a full-fledged type theory for cost analysis **catt** in Section 2.1. As a computational type theory built on top of an operational semantics, **catt** adopts a uniform cost model in the sense described in **(1.1.5∗1)**.

(1.3∗5) On the other hand, we may begin with the synthetic perspective by viewing type theories as generalized algebraic theories. Consequently, the meaning of type connectives is not explained via more primitive means (*e.g.* through computation as in computational type theories); rather, one views the rules of type theory as *synthetically* characterizing the desired properties of various type connectives. In constrast to the analytic perspective **(1.3∗4)**, the algebraic view of type theory naturally supports a compositional equational theory, which is also compatible with the integration of cost structure using either the monadic approach **(1.1.2∗4)** or the more fine-grained CBPV presentation (Section 1.1.3).

In Section 2.2 we build on the dependent CBPV calculus of Pédrot and Tabareau and develop a cost-aware logical framework **calf** that implements cost structure as a computational effect. As we discuss in **(2.2∗2)** **calf** supports a non-uniform cost model; I conjecture that **calf** is expressive enough to embed uniform cost models in the sense of **(1.1.5∗3)** and propose to use **calf** to prove internal adequacy theorems in Section 3.2, leading to the view of **calf** as a framework for *cost-aware denotational semantics.* As foreshadowed in **(1.2∗3)**, extensional equality is expressed in **calf** by means of an *extensional phase* that is directly analogous to the notion of the *static phase* in the theory of ML modules; as a consequence **calf** evinces a new phase distinction of *intension/cost* and *extension/behavior.*

1.4. THESIS STATEMENT

> *Emerging from the synthesis of computational effects and phase-distinct programming in dependent type theory is a cost-aware logical framework **calf** that furnishes a language for equational reasoning of both cost/intensional and behavioral/extensional structures. A framework of this kind unifies three previously distinct roles: a cost-aware programming language, a mathematical domain for algorithm analysis, and a metalanguage for cost-aware denotational semantics. Consequently, **calf** is a natural mathematical space in which to study cost-aware computation*

at all scales, ranging from algorithm specification and verification to the cost semantics of programming languages.

CHAPTER 2

ANALYTIC AND SYNTHETIC COST STRUCTURES

2.1. ANALYSIS: COST-AWARE TYPE THEORY

2.1.1. Computational type theories.

(2.1.1∗1) The fundamental ethos behind the computational type theories of Martin-Löf [Mar79] and Constable et al. [Con+86] is the idea that the meaning of type connectives is *defined* via computation. It is in this sense that computational type theories may be characterized as analytic theories: as the object of study, type structure arises directly as an extrinsic property of the prior notion of computation. The explanation of type-theoretic constructions in terms of computation is known as Martin-Löf's *meaning explanation* of type theory.

(2.1.1∗2) In the context of computational type theory the meaning explanation is built on top of a programming language equipped with an operational semantics (for instance as described in **(1.1.1∗1)**), and the judgmental structure reflects programming concepts such as when a computation represents a specification (*i.e.* a type) and when a computation satisfies a given specification.

(2.1.1∗3) According to the meaning explanation, to know that a judgment holds is to know that there is a direct proof of the judgment. Applied in a computational setting, this means that *e.g.* a program is a specification just when it *computes* to a *canonical* specification, *i.e.* a canonical type. The idea of canonical objects has a special status in the cost-aware setting because they correspond to *values* or programs (and specifications) with no associated cost.

(2.1.1∗4) Martin-Löf's meaning explanation of type theory is structured around four forms of *judgment*: A is a type, M is a term of A, A and A' are equal types, and M and M' are equal terms of A, displayed as follows using shorthand notations:

$$\begin{array}{ll} A\ \mathsf{type} & M \in A \\ A \doteq A' & M \doteq M' \in A \end{array}$$

Examining the meaning-theoretic structure of each reveals the central importance of computation. For instance, to have a proof of the typehood judgment A type is know that A *computes* to a *canonical type* A_0, *i.e.* a partial equivalence relation on programs. Furthermore, presupposing that A type (so that A evaluates to a canonical type A_0), to have a proof of the membership judgment $M \in A$ is to know that M computes to a canonical value V and that V is related to itself by the partial equivalence relation associated with the canonical type A_0. The fact that canonical types are associated with PERs allows us to explain the *equality* of terms at a given type.

(2.1.1∗5) The definitional principle associated with the meaning explanation of computational type theories is a well-known instance of *induction-recursion*, which is a principle that allows one to define an inductive structure and a recursive function out of that structure simultaneously. Here, one may render the judgment of typehood and membership as a pair (U, T) defined by induction-recursion Dybjer and Setzer [DS04] in which $U : \mathsf{Tm} \to \mathsf{Set}$ classifies the set of evidence for establishing typehood and $T : (e : \mathsf{Tm}) \to U(e) \to \mathsf{Tm} \to \mathsf{Set}$ classifies the set of evidence for establishing membership given the computability data of a type. The purpose of induction-recursion is to allow the *simultaneous* definition of the inductive set $U(e)$ and the recursive function $T(e)$ out of $U(e)$ for a given term e.

(2.1.1∗6) On the other hand researchers in the **Nuprl** school justify Martin-Löf's meaning explanation for computational type theories by various forms of fixed-point constructions over a "lookup table" [Har92; All87]. The idea is to encode the judgmental structure of a type theory as a relation τ on $\mathsf{Tm} \times \mathsf{Tm} \times \mathcal{P}(\mathsf{Tm} \times \mathsf{Tm})$ so that $\tau(A, A', \phi)$ roughly means that A and A' are equal canonical types equipped with the membership relation ϕ.

Because every powerset with the subset ordering has the structure of a complete lattice, one may use the Knaster-Tarski theorem to construct a type system as the fixed-point of a monotone function on relations. For instance, a type theory with Π types and the natural numbers may be defined as the fixed-point of the following monotone operator Φ (following the notation of Angiuli [Ang19]):

$$\begin{aligned}
&\mathrm{Nat} : \mathcal{P}(\mathsf{Tm} \times \mathsf{Tm} \times \mathcal{P}(\mathsf{Tm} \times \mathsf{Tm})) \to \mathcal{P}(\mathsf{Tm} \times \mathsf{Tm} \times \mathcal{P}(\mathsf{Tm} \times \mathsf{Tm}))\\
&\mathrm{Nat}(\tau) = \{\, (\mathtt{nat}, \mathtt{nat}, \mu\alpha.\,\{\, (\mathsf{zero}, \mathsf{zero}) \,\} \cup \{\, (\mathsf{suc}(M), \mathsf{suc}(M')) \mid M \sim M' \in \alpha \,\}) \,\}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Fun} : \mathcal{P}(\mathsf{Tm} \times \mathsf{Tm} \times \mathcal{P}(\mathsf{Tm} \times \mathsf{Tm})) \to \mathcal{P}(\mathsf{Tm} \times \mathsf{Tm} \times \mathcal{P}(\mathsf{Tm} \times \mathsf{Tm}))\\
&\mathrm{Fun}(\tau) = \{\, (\Pi(A, a.B), \Pi(A', a.B'), \phi) \mid\\
&\qquad \exists \alpha.\, \tau \vDash A \sim A' \downarrow \alpha\\
&\qquad \wedge\, \beta.\, \tau \vDash \alpha \rhd B \sim B' \downarrow \beta\\
&\qquad \wedge\, \phi = \{\, (\lambda x.\, M, \lambda x.\, M') \mid \alpha \rhd M \sim M' \in \beta \,\}\}
\end{aligned}$$

$$\Phi(\tau) = \tau \cup \mathrm{Nat}(\tau) \cup \mathrm{Fun}(\tau)$$

A given relation τ is a *type system* when it satisfies some intuitive coherence conditions.[1] Given a type system τ, one may build a computational type theory by defining all the required judgmental structure relative to τ. For instance, to know that $A\ \mathsf{type}$ is to have that $\tau \vDash \exists \alpha.\, A \sim A \downarrow \alpha$, *i.e.* A computes to a canonical type that is assigned some membership relation α.

(2.1.1∗7) The construction of a type system given in **(2.1.1∗6)** may seemed involved, but the general method scales to model more complicated computational phenomena such as guarded recursion [SH18] and (cartesian) cubical structures [Ang19].

2.1.2. Meaning of cost structure.

(2.1.2∗1) In this section we recount the essential ideas behind **catt** (cost-aware type theory) [NH20], a computational type theory in the **Nuprl** tradition that furnishes a formal framework for developing *cost-aware* program verification.

(2.1.2∗2) *Cost structures, analytically.* Because both Martin-Löf's meaning explanation and the construction of computational type theories are built on top of operational semantics, the cost structure of programs was already present and lurking behind the scene in every judgment of type theory. For instance, implicit in every proof of the judgment $A\ \mathsf{type}$ is the knowledge that A computes to a canonical type A_0 *while incurring some cost c.* The essential idea of **catt** is to make the presence of cost structure explicit and replay the constructions in Section 2.1.1 to obtain a *cost-aware* computational type theory.

[1]See Angiuli [Ang19] and Niu and Harper [NH20]

(2.1.2∗3) *Cost structure over an operational semantics.* Given any small-step structural operational semantics $\mapsto$, we may induce a cost semantics on programs in which a program M has cost $c : \mathbb{N}$ whenever $M \mapsto^c V$ for a value V. Cost bounds are defined analogously: the cost of a program M is bounded by $c' : \mathbb{N}$ whenever $M \mapsto^c V$, V is a value, and $c \leq c'$.

(2.1.2∗4) Although **(2.1.2∗3)** provides a good notion of (metatheoretic) cost structure, one needs to be able to speak about cost structure internally. This means that cost structure should be given by programs and that we need to judge when a given program is a cost bound.

(2.1.2∗5) Cost bounds are just natural numbers, and so a program P is a cost bound just when $P \in \texttt{nat}$. As we will see in **(2.1.2∗6)** it will be important that `nat` classifies *numerals* rather than arbitrary computations.

(2.1.2∗6) *Cost-aware judgments in **catt**.* To expose the cost structure of programs, we define two new forms of judgments that express the idea that programs satisfy specifications within the constrains of a cost bound:

$$M \in A\ [P] \qquad M \doteq M' \in A\ [P]$$

We may endow these judgment forms with meaning-theoretic structure: to know that $M \in A\ [P]$ is to know that (presupposing P is cost bound, *i.e.* $P \in \texttt{nat}$)

1. $M \Downarrow^c V$
2. $P \Downarrow \overline{p}$
3. $c \leq p$
4. $V \in A$

Similarly, the cost-aware equality at a type is defined by requiring that both programs are bounded by the given cost bound (and satisfies the given specification).

Note that because we have required `nat` to classify strictly the numerals, it is always possible to "read off" the underlying actual natural number by evaluating the cost bound.

2.1.3. Cost-aware type theory.

(2.1.3∗1) *The programming language **PL**.* Underlying **catt** is the programming language **PL**, essentially a version of PCF [Plo77] equipped with a call-by-value operational semantics, governed by two standard relations over untyped terms $\mapsto\subseteq \mathsf{Tm}\times\mathsf{Tm}$ and $\mathsf{val}\subseteq\mathsf{Tm}$. Observe that in this setting canonical proofs/members in the sense of the meaning explanation are drawn from the the subset val of values or terminal programs.

(2.1.3∗2) The fact that we consider a call-by-value semantics is somewhat forced by considering the semantics of *open* judgments. Without drastically altering the structure of hypothetical judgments, it is not possible to characterize the cost bound of an open term if one is allowed to substitute arbitrary computations for variables. For instance, consider an open term $x : A \gg N \in B$, and write $[\![A]\!] := \{ M \mid M \in A \}$ for the closing instances for the free variable $x : A$. We see that the cost of N cannot be characterized by a function $[\![A]\!] \to \mathsf{nat}$: without knowing the cost of a given computation $M \in A$, we cannot in general determine the cost of the composition $[M/x]N \in [M/x]B$.

(2.1.3∗3) Consequently, the notion of a canonical proof or canonical member as foreshadowed in **(2.1.1∗3)** is distinguished in **catt** because they span the domain of quantification for open or hypothetical judgments. To distinguish a canonical member from an arbitrary one, we write $V \in_0 A$ for $V \in A$ whenever V is a value.

(2.1.3∗4) In the absence of a modal account of computations in the style of Harper [Har20], one may only form fixed-points at function types in **PL**, which has the following operational behavior when applied to a value:

$$\frac{V\ \mathsf{val}}{\mathtt{fun}(x.y.M)V \mapsto [\mathtt{fun}(x.y.M)/y, V/x]M}$$

(2.1.3∗5) *Cost-aware function types.* Governing the recursive functions is the fundamental unit of cost refinement in **catt**, the cost-aware function type written as $(a : A) \to B\ [P]$. Intuitively, a canonical member of the specification $(a : A) \to B\ [P]$ is a (recursive) function $\mathtt{fun}(x.y.M)$ such that any application instance is cost bounded in the sense described in **(2.1.2∗6)**. Explicitly, this means we would expect that $f \in (a : A) \to B\ [P]$ whenever $[\mathtt{fun}(x.y.M)/y, V/x]M \in [V/a]B\ [[V/a]P]$ for all $V \in_0 A$. Crucially the cost bound P is allowed to vary in the input, thereby forcing us to associate judgmental structure to the notion of a cost bound.

(2.1.3∗6) *Hypothetical judgments.* The general structure the cost-aware function type connective internalizes is that of the hypothetical cost-aware judgments:

$$\Gamma \gg M \in A\ [P] \qquad \Gamma \gg M \doteq M' \in A\ [P]$$

In the judgments above the cost bound is allowed to reference the entire context. As mentioned in **(2.1.3∗2)** the hypothetical judgments quantify over substitution instances valued in (equal) canonical members, and as usual the meaning of such judgments is given by universal quantification over equal substitution instances for the context.

(2.1.3∗7) *Constructing the type system.* As described in **(2.1.1∗6)**, the judgments of a computational type theory is defined relative to a type system τ. The type system relative to which judgments of **catt** differs from the fixed-point construction of standard computational type theories (for instance the "Idealized Nuprl" of Angiuli [Ang19]) in two places. First, as discussed in **(2.1.2∗6)** we have to arrange the construction so that the type of natural numbers classify numerals. Concretely, this just means swapping out the member relation out for ω, the diagonal set over the numerals:

$$\begin{aligned}
\mathrm{Nat} &: \mathcal{P}(\mathsf{Val} \times \mathsf{Val} \times \mathcal{P}(\mathsf{Val} \times \mathsf{Val})) \to \mathcal{P}(\mathsf{Val} \times \mathsf{Val} \times \mathcal{P}(\mathsf{Val} \times \mathsf{Val})) \\
\mathrm{Nat}(\tau) &= \{ (\mathtt{nat}, \mathtt{nat}, \omega) \}
\end{aligned}$$

Secondly, we have to define a new (monotone) operator on type systems that adds in the cost-aware function types **(2.1.3∗5)**:

$$\begin{aligned}
\mathrm{Fun} &: \mathcal{P}(\mathsf{Val} \times \mathsf{Val} \times \mathcal{P}(\mathsf{Val} \times \mathsf{Val})) \to \mathcal{P}(\mathsf{Val} \times \mathsf{Val} \times \mathcal{P}(\mathsf{Val} \times \mathsf{Val})) \\
\mathrm{Fun}(\tau) &= \{ ((a : A) \to B[P], (a : A') \to B'[P'], \phi) \mid \\
&\quad \exists \alpha.\, \tau \vDash A \sim A' \downarrow \alpha \\
&\quad \wedge \beta.\, \tau \vDash \alpha \rhd B \sim B' \downarrow \beta \\
&\quad \wedge \alpha \rhd P \sim P' \in \omega \\
&\quad \wedge \phi = \{ (\mathtt{fun}(x.y.M), \mathtt{fun}(x.y.M')) \mid \alpha \rhd [\mathtt{fun}(x.y.M)/y]M \sim [\mathtt{fun}(x.y.M')/y]M' \in \beta[P] \}\}
\end{aligned}$$

In the above the notation $\alpha \rhd M \sim M' \in \beta[P]$ expresses a functionality condition, which states that given inputs V, V' such that $\alpha(V, V')$, $[V/a]M$ and $[V'/a]M'$ are behaviorally equivalent up the specification $\beta_{V,V'}$ and cost bound $P[V/a]$.

(2.1.3∗8) What is noteworthy of the construction of cost-aware functions outlined in **(2.1.3∗7)** is the fact that the functionality of recursive functions $\mathtt{fun}(x.y.M)$ is stated in such a way that only computability of the first argument is assumed and there are *no* assumptions about the computability of the recursive argument. Compared to the syntactic typing discipline of recursive functions, the membership relation assigned to cost-aware function types appears to be too strong to be useful in practice. However as we will explore in Section 2.1.4, one may exploit cost structure to *derive* a useful rule about cost-aware function types that simultaneously establishes membership and cost refinement.

2.1.4. Cost refinement rules.

(2.1.4*1) Confronted with the lack of useful computability data **(2.1.3*8)**, we realized that the additional structure of the cost bound provides enough information for us to *derive* strong reasoning principles for the cost-aware function type. First, we see that given a type $(a : A) \to B\ [P]$, the cost bound P induces an ordering $\prec$ on the arguments defined by $V \prec_P W :\equiv [V/a]P < [W/a]P$. The idea is that in order for a recursive function $\texttt{fun}(x.y.M)$ to adhere to a cost bound P on a given argument V, the associated recursive calls can only be made on arguments V' such that $V' \prec_P V$. In other words, we may assume that the recursive function binding behaves correctly *for arguments less than the current argument on the cost ordering* $\prec$. Because function application consumes one step according to the operational semantics, this is the strongest assumption one may have about the recursive binding.

(2.1.4*2) This leads to a principle one may refer to as *induction on cost* or *cost bound induction*, and a variation of this idea also appears later in the encoding of (total) general-recursive programs in **calf**.

(2.1.4*3) Going back to the question of how one can prove that a function inhabits a cost-aware function type, a direct reading of the semantics of cost-aware function types would lead to the following introduction rule:

$$\frac{\begin{array}{c} A \doteq A' \ \mathsf{type} \\ a : A \gg B \doteq B' \ \mathsf{type} \\ a : A \gg P \doteq P' \in \texttt{nat} \\ a : A \gg [\texttt{fun}(x.y.M)/y]M \doteq [\texttt{fun}(x.y.M')/y]M' \in B\ [P] \end{array}}{\texttt{fun}(x.y.M) \doteq \texttt{fun}(x.y.M') \in_0 (a : A) \to B\ [P]}$$

In general the rule displayed above is not very useful as an introduction rule because nothing is known about the recursive binding. What cost bound induction buys us is the additional assumption in the last premise stating that the recursive argument satisfies the stated specification *for arguments that induce strictly less cost.* This is rendered as the the following lemma:

$$\dfrac{\begin{array}{c} A \doteq A' \ \mathsf{type} \\ a : A \gg B \doteq B' \ \mathsf{type} \\ a : A \gg P \doteq P' \in \mathtt{nat} \\ a : A, f : (a : \{\, a' \mid a' \prec_P a \,\}) \to B\ [P] \gg M \doteq M' \in B\ [P] \end{array}}{\mathtt{fun}(a.f.M) \doteq \mathtt{fun}(a.f.M') \in_0 (a : A) \to B\ [P]}\ (\Pi_{◔}{-}I)$$

Because the extra data on the admissible arguments is a proposition, it is stated using the subset connective in the above, but a presentation using Σ-types is also possible.

(2.1.4∗4) The cost refinement $\Pi_{◔} - \mathrm{I}$ reveals a serendipitous principle distinctly available in the cost-aware setting:

> Cost structure realizes efficient algorithms, and efficient algorithms has good cost structure.

Here I say “to realize” to mean to demonstrate or prove the existence of a realizer of a given cost-aware specification. Because one is making explicit the cost structure of programs, the induction principle associated with recursive functions may be strengthened appropriately by limiting the admissible arguments to the recursive call.

The fact that cost structure induces a strengthened refinement rule is an instance of the difference between *safety* and *liveness* properties. In the context of **catt**, while it may be difficult to show that a program *merely terminates* (a liveness property), it is much easier to check whether or not it terminates with a given cost bound (a safety property). In a purely behavioral setting one has to decide whether it is worth giving an *explicit description* of the data inherit in termination proofs. In contrast the *raison d'être* of **catt** forces us to always be explicit about the cost structure in the specification, which is in turn used to show that candidate programs satisfy cost-aware specifications.

(2.1.4∗5) The nature of program verification in **catt** is characteristic of computational type theories in that it is based on an *open-ended* collection of proof refinement lemmas. Aside from the new cost refinement rule $\Pi_{◔} - \mathrm{I}$, **catt** also admits the standard syntax-directed lemmas governing the formation, introduction, and elimination of the standard connectives including `nat`, Σ, subset, and extensional equality. Here another workhorse lemma is the elimination rule for `nat`, which states that the cost of a case analysis is another case analysis:

$$\frac{\begin{array}{c}\cdots \\ V \doteq V' \in_0 \texttt{nat} \\ p : \mathsf{eq}_{\texttt{nat}}(\mathsf{zero}, M) \gg M_0 \doteq M_0' \in [\mathsf{zero}/a]A\ [P_0] \\ a : \texttt{nat}, p : \mathsf{eq}_{\texttt{nat}}(\mathsf{suc}(a), V) \gg M_1 \doteq M_1' \in [\mathsf{suc}(a)/a]A\ [P_1]\end{array}}{\texttt{ifz}(V)\{M_0; a.M_1\} \doteq \texttt{ifz}(V')\{M_0'; a.M_1'\} \in [V/a]A\ [\mathsf{ifz}_V(\mathsf{suc}(P_0); a; \mathsf{suc}(P_1))]}$$

In the above we have omitted the expected typing presuppositions.

2.1.5. catt as a cost analysis framework.

(2.1.5∗1) Using the rules highlighted in **(2.1.3∗5)** and **(2.1.4∗5)**, I was able to verify (by hand) a crude upper bound on Euclid's algorithm for gcd. Although this was a victory for us and demonstrated that one could use **catt** to do cost analysis in *theory*, Euclid's algorithm is likely the upper limits of what one can manage on paper, and it was clear that we needed a mechanization of **catt** to handle bigger case studies.

(2.1.5∗2) Around January of 2020 I begin mechanizing the core definitions and construction of **catt**. After the better part of a year I formally verified $\Pi_{◔} - \mathrm{I}$, the distinguish rule in **catt** for typing recursive functions. However this is basically the only rule I verified, and at this point I decided that formalizing *any* case study would be too painful to carry out in the current formulation of **catt**.

(2.1.5∗3) What makes the practice of verification difficult in a by-the-books implementation of **catt** is *not* that there are no untyped computational principles à la Howe [How89]. Indeed Niu and Harper [NH20] prove open head expansion for the ordinary membership and typehood judgments; as far as the behavioral fragment is concerned, it seems possible to extend this to an approximation relation as in Sterling and Harper [SH18].

(2.1.5∗4) The predictable problem is the fact that cost reasoning is *sub*equational, thus ordinary equations do not apply directly to cost-aware judgments. What is insidious is the interpretation of variables in **catt**. The fact that open judgments only quantify over substitutions valued in values has some uncomfortable effects on ordinary judgments such as $\Gamma \gg A$ type. For instance, consider the judgment $a : A \gg B$ type. Although any equal values $V \doteq V' \in_0 A$ give rise to equal specifications $[V/a]B \doteq [V'/a]B$ type, this does *not* hold for arbitrary equal instantiations $M \doteq M' \in A$. In particular, it is not the case that $\langle M/a \rangle\, B \doteq [V/a]B$ type given that

$M \doteq V \in A$ and $V \in \mathsf{val}$. As mentioned in **(1∗2)**, type families should respect *extensional* equality of their indices. However, the interpretation of variables as values in **catt** results in a nonstandard version of type families and complicates the development of ordinary mathematics.

(2.1.5∗5) Compounded with the general engineering efforts required to make *implementations* of computational type theories in the **Nuprl** style ergonomic and practical to use, by the latter half of 2020 I started thinking about cost analysis in type theory from a different angle, this time with mechanization at the fore.

2.2. SYNTHESIS: A COST-AWARE LOGICAL FRAMEWORK

(2.2∗1) In this section I discuss the development of **calf** (cost-aware logical framework), a dependent type theory with an intrinsic notion of cost structure and cost-aware programs. At a high-level, **calf** resolves many of the problems we encountered with **catt** and traditional accounts of cost structure in type theory by synthesizing several recent techniques: dependent types with computational effects [PT19], synthetic phase distinction/noninterference [SH21], and cost-aware programming à la Niu and Harper [NH20].

(2.2∗2) As mentioned in **(1.3∗5)**, in contrast to **catt**, we embrace an *abstract* notion of cost in **calf** that is not necessarily tied to an operational semantics. Following actual practice of algorithm analysis, we do not associate a particular cost semantics to **calf** itself but instead promote the use of **calf** as a cost-aware metalanguage for expressing algorithm-specific/non-uniform cost models in the sense of **(1.1.5∗1)**. We will discuss the connection between the traditional formulation of cost given by an operational semantics and the non-uniform cost model of **calf** in Section 3.2.

(2.2∗3) *Ab initio* we have set out to ensure that **calf** can be implemented on a computer for the mechanization of large-scale verifications. This has been carried out by the Cost Lab Refinement team (Jon Sterling, Harrison Grodin, and myself). All the case studies we discuss in Section 2.2.9 are mechanically verified theorems in the Agda proof assistant.[2]

[2]The source code is available at `https://github.com/jonsterling/agda-calf`.

2.2.1. Cost as a computational effect.

(2.2.1∗1) As we discussed in **(1.1.2∗2)**, while cost monads are unproblematic as a denotational semantics for cost-aware programs, the semantic domain $\mathbb{C} \times -$ is not itself suitable for cost-aware programming because too much detail about the implementation of cost structure is revealed. However, we can construct a suitable syntax for cost-aware programming by considering the *call-by-push-value* structure [Lev04] induced by the Eilenberg-Moore category associated with the cost monad. Recall from our discussion in Section 1.1.3 that the language of CBPV is based around the dichotomy of values and computations at the level of both terms and types. In the Eilenberg-Moore model of CBPV, value types correspond to ordinary sets while computation types are interpreted as algebras over the given monad. Consequently value types and computation types are bridged by a free-forgetful adjunction $\mathsf{F} \dashv \mathsf{U}$ in which the left adjoint sends a set A to the free algebra $\mathbb{C} \times A$ and the right adjoint forgets the algebra structure.

(2.2.1∗2) *A cost-aware CBPV language.* The semantic situation in **(2.2.1∗1)** then justifies a new computation effect $\mathsf{step}^c(e)$ that incurs a cost $c : \mathbb{C}$ to the computation $e : X$, complete with the expected coherence law $\mathsf{step}^{c+d}(e) = \mathsf{step}^c(\mathsf{step}^d(e))$ that combines multiple step's using the monoid structure on $\mathbb{C}$. Moreover we may reify the free functor F as a type constructor with the following introduction and elimination rules::

$$\begin{aligned} \mathsf{ret} &: \{A\}\ A \to \mathsf{F}(A) \\ \mathsf{bind} &: \{A, X\}\ \mathsf{F}(A) \to (A \to X) \to X \end{aligned}$$

The interaction between return, sequential composition, and the cost effect is characterized by the following equations:

$$\begin{aligned} \mathsf{bind/ret} &: \mathsf{bind}(\mathsf{ret}(a); f) = f(a) \\ \mathsf{bind/step} &: \mathsf{bind}(\mathsf{step}^c(e); f) = \mathsf{step}^c(\mathsf{bind}(e; f)) \end{aligned}$$

(2.2.1∗3) *Equational theory for cost analysis.* We observe that the equational theory of CBPV furnishes a theory for cost analysis: a computation $e : \mathsf{F}(A)$ has cost $c : \mathbb{C}$ whenever $e = \mathsf{step}^c(\mathsf{ret}(a))$ for some value $a : A$. This *cost refinement* is captured by the following internal predicate:

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

Similarly, we may also define the refinement of *cost bounds* that allows for upper bounds on cost. Cost analyses in **calf** revolve around cost refinements such as

$\mathsf{hasCost}$ and syntax-direct refinement lemmas **(2.2.6∗3)** that express the cost bound of a computation based on cost bounds of its constituent sub-computations.

Although program verification in **calf** looks superficially similar to that of **catt**, the actual experience of proving cost bounds in **calf** critically differs from **catt** because one can apply *local equational reasoning*. In other words one may use an equation in any context whatsoever in **calf**. In contrast the cost-sensitive theorems of **catt** are sub-equational. Moreover, **catt** suffers from a deficiency of equations stemming from the coarse-grained type structure induced by the underlying call-by-value language.

(2.2.1∗4) *Effects and dependency.* A subtle point is how to make sense of a *dependent* bind operation. Consider a type family $X : A \to \mathsf{U}$ computations $e : \mathsf{F}(A)$ and $f : (a : A) \to X(a)$. What should be the type of the sequential composition of e and f? One answer is given by Pédrot and Tabareau [PT19], who develop a dependent version of the classic CBPV calculus dubbed **∂cbpv** in which there are *three* sequencing operations. Aside from the ordinary non-dependent bind, one has the dependent sequencing dbind and type-level sequencing tbind:

$$\begin{aligned} &\mathsf{tbind} : \{A\}\ \mathsf{F}(A) \to (A \to \mathsf{U}) \to \mathsf{U} \\ &\mathsf{dbind} : \{A, X\}\ (e : \mathsf{F}(A)) \to ((a : A) \to X(a)) \to \mathsf{tbind}(e; X) \end{aligned}$$

Building on this work we have developed **calf** as an extension to **∂cbpv**.

2.2.2. Phase distinction revisited: intension *vs.* extension.

(2.2.2∗1) The fragment of **calf** described so far is sufficient for *expressing* the notion of cost refinements. However as I have argued in **(1.1∗5)**, to *prove* cost refinements, one must also be able to express behavioral/extensional specifications.

(2.2.2∗2) In fact the intensional aspects of programs (*e.g.* cost) is already well-studied in the structural proof theory and modal type theory community under the guise of staged computation [DP99]. In this context the necessity modality $\Box$ is used to capture the idea of staged computation; in particular, Davies and Pfenning [DP99] notes that one may think of $\Box A$ as the type that classifies “codes” or representations of terms of type A.

In his dissertation Kavvos carries out a detail investigation of a intensional type theory (intensional PCF) based on this interpretation of the necessity modality. Unfortunately Kavvos’s work ultimately shows that truly intensional operations may only be effected on *closed terms*, which greatly complicates the integration of dependent types and generally erodes the benefits of working type-theoretically.

(2.2.2∗3) Around the beginning of 2021, Sterling observed that the problem I encountered when trying to introduce intensional structures in dependent type theory may be resolved using the same tools that Sterling and Harper [SH21] developed to objectify and study the ML module system. The central difficulty of prior work on expressing intensional structures through the necessity modality is that judgmental equalities are *removed* in certain contexts; Sterling suggests that the situation may be dually and more beneficially viewed as the *addition* of equations that express extensional/behavior equality in certain contexts, which is analogous to the erasure of dynamic components of a module structure in the *static phase* in the theory of module systems. This led us to consider a new *phase distinction* in the context of cost analysis, which is the phase distinction of *intension* and *extension*.

2.2.3. Intension *vs.* extension: a new phase distinction.

(2.2.3∗1) In the tradition of ML module systems, a module is a composite structure that has both a dynamic/runtime component and a static/compile time component. The functional dependency of modules is managed through the notion of a module functor, which is just a function between the corresponding module signatures subject to the restriction that the static components may not depend on the dynamic component coming from the domain signature. This restriction constitutes the original phase distinction of the *static* and *dynamic*.

(2.2.3∗2) With a bit of squinting we see that the phase distinction is exactly the safety property defined in **(1.1.2∗2)**; we just make the following substitutions: *dynamic* → *intension*, *static* → *extension*, and *module functor* → *cost-aware function*.

2.2.4. The language of phase distinctions.

(2.2.4∗1) As mentioned in **(2.2.2∗3)**, Sterling and Harper promotes a type-theoretic interpretation of the theory of module systems. In particular *op. cit.* presents a synthetic reconstruction of the phase distinction of the static and dynamic. Briefly, one may carve out the static phase of the module system by means of distinguished proposition $\P_{\mathsf{St}} : \Omega$ called the "static open"[3]; dynamic components of a module are trivialized whenever a proof $u : \P_{\mathsf{St}}$ is present in the context.

(2.2.4∗2) In the setting of cost-aware programs we may make an entirely analogous move and implement the *extensional phase* by means of a distinguished proposition

[3]An open in the sense of topos theory is a proof-irrelevant proposition in the internal type theory of a topos.

$\P_{\mathsf{E}} : \Omega$ called the extensional open. Whenever a proof of $\P_{\mathsf{E}}$ is available the cost structure of programs is rendered trivial; in light of the effectful conception of cost structure in **(2.2.1∗2)**, this is expressed by the following rule:

$$\mathsf{step}/\P_{\mathsf{E}} : \{X, e\}\ (u : \P_{\mathsf{E}}) \to \mathsf{step}^c(e) = e$$

Consequently the extensional part of any type may be extracted by the *extensional modality* $\bigcirc$, which is defined by exponential with the extensional open: $\bigcirc A := (u : \P_{\mathsf{E}}) \to A$. For instance, we may prove the extensional-modal equation $\bigcirc(\mathit{insertionSort} = \mathit{mergeSort})$ and yet still be able to distinguish their costs *outside* the extensional phase. As I will discuss in **(2.2.6∗1)**, in **calf** one also reasons about inequalities between cost bounds in the extensional phase: as we have observed in **(1.1∗5)** cost analysis is dependent on extensional/behavioral properties of the programs involved.

(2.2.4∗3) Conversely, we may use the *intensional modality* to seal away cost structure that should be erased in the extensional phase. In type theory, the intensional modality may be defined as the following quotient inductive type:

$$\begin{aligned}
&\textbf{data } \bullet A \textbf{ where}\\
&\quad \eta_\bullet : A \to \bullet A\\
&\quad\ \ * : \P_{\mathsf{E}} \to \bullet A\\
&\quad\ \ \_ : \Pi a : A.\, \Pi u : \P_{\mathsf{E}}.\eta_\bullet(a) = *(u)
\end{aligned}$$

In categorical language, the above may be seen as the pushout of the projections of $A \times \P_{\mathsf{E}}$. It is a bit more difficult to visualize the meaning of the intensional modality, but one can imagine $\bullet A$ as identical to A except that it is trivial inside the extensional phase, *i.e.* $\bigcirc\bullet A \cong 1$. A useful way to internalize this fact is the phrase "the extension part of the intensional part is trivial". In **(3.2.3∗2)** we will use the intensional modality to state a cost-aware version of the Plotkin-type adequacy theorem.

2.2.5. calf: a cost-aware logical framework.

(2.2.5∗1) *Presenting type theory using logical frameworks.* The fragments of **calf** highlighted in Section 2.2.1 and Section 2.2.4 are pieced together into a type theory as a signature in the logical framework of locally closed cartesian categories. This departure from traditional presentations of type theory as derivations built over raw terms is promoted by several recent threads of work [Uem19; SH21; GS20]. As argued by Sterling [Ste21], the study of type theories *qua* mathematical objects in structured categories allows one to dispense with many technical difficulties when defining models of "hand-baked" presentations of type theory. Indeed we will see in Section 2.2.10 that this view of type theories enables one to easily define models of **calf**.

(2.2.5∗2) Concretely, we work in a logical framework with a universe of judgments **Jdg** closed under dependent product, dependent sum, and extensional equality. An object theory (*e.g.* **calf**) is specified as follows:

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

The core constructs of **calf** are displayed in Fig. 2.1; the complete definition may be found in Appendix .1.

Note that **calf** is parameterized in an arbitrary cost monoid $(\mathbb{C}, 0, +, \leq)$ and the predicate $\mathsf{isOrderedMonoid}$ encapsulates the ordered monoid laws. This allows us to encode different abstract notions of cost structure; in particular we may instantiate $\mathbb{C}$ as the parallel cost monoid to account for the parallel complexity of programs (Section 2.2.9).

2.2.6. Interactive cost refinement in calf.

(2.2.6∗1) *Extensional cost bounds.* As a first attempt to generalize the $\mathsf{hasCost}$ refinement from **(2.2.1∗3)**, we may conjecture that a computation $e : \mathsf{tm}^{\ominus}(\mathsf{F}(A))$ is bounded by $c : \mathbb{C}$ if $e =_{\mathsf{tm}^{\ominus}(\mathsf{F}(A))} \mathsf{step}^{c'}(\mathsf{ret}(a))$ for some $c' \leq c$ and $a : \mathsf{tm}^{+}(A)$. While this is a perfectly sensible definition, our investigations suggest it is more natural to replace ordinary inequality $\leq$ with the *extensional inequality* $\bigcirc(c' \leq c)$. The use of the extensional inequality in the $\mathsf{IsBounded}$ refinement reflects the intuition that "costs don't have cost". More importantly, this arrangement grants one access to the extensional fragment and the *extensional* properties therein when proving cost refinements, which is essential for analyses of algorithms that depend on behavioral invariants of data structures. For instance, the cost analysis of insertion sort depends on knowing the invariant that sorting preserves length, a fact that follows from the correctness of sorting that is readily available in the extensional phase.

$$
\begin{aligned}
\mathbb{C} &: \mathbf{Jdg} \\
0 &: \mathbb{C} \\
+ &: \mathbb{C} \to \mathbb{C} \to \mathbb{C} \\
\leq &: \mathbb{C} \to \mathbb{C} \to \mathbf{Jdg} \\
\mathsf{costMon} &: \mathsf{isOrderedMonoid}(\mathbb{C}, 0, +, \leq) \\
\mathsf{step} &: \{X : \mathsf{tp}^{\ominus}\}\ \mathbb{C} \to \mathsf{tm}^{\ominus}(X) \to \mathsf{tm}^{\ominus}(X) \\
\mathsf{step}_0 &: \{X, e\}\ \mathsf{step}^0(e) = e \\
\mathsf{step}_+ &: \{X, e, c_1, c_2\} \\
&\quad \mathsf{step}^{c_1}(\mathsf{step}^{c_2}(e)) = \mathsf{step}^{c_1+c_2}(e)
\end{aligned}
\qquad
\begin{aligned}
\mathsf{tp}^+ &: \mathbf{Jdg} \\
\mathsf{tm}^+ &: \mathsf{tp}^+ \to \mathbf{Jdg} \\
\mathsf{U} &: \mathsf{tp}^{\ominus} \to \mathsf{tp}^+ \\
\mathsf{F} &: \mathsf{tp}^+ \to \mathsf{tp}^{\ominus} \\
\mathsf{tm}^{\ominus}(X) &:= \mathsf{tm}^+(\mathsf{U}(X)) \\
\mathsf{ret} &: (A : \mathsf{tp}^+, a : \mathsf{tm}^+(A)) \to \mathsf{tm}^{\ominus}(\mathsf{F}(A)) \\
\mathsf{bind} &: \{A : \mathsf{tp}^+, X : \mathsf{tp}^{\ominus}\}\ \mathsf{tm}^{\ominus}(\mathsf{F}(A)) \to \\
&\quad (\mathsf{tm}^+(A) \to \mathsf{tm}^{\ominus}(X)) \to \mathsf{tm}^{\ominus}(X)
\end{aligned}
$$

$$
\begin{aligned}
¶_{\mathsf{E}} &: \mathbf{Jdg} \\
¶_{\mathsf{E}}/\mathsf{uni} &: \{u, v : ¶_{\mathsf{E}}\}\ u = v \\
\\
\bigcirc\mathcal{J} &:= ¶_{\mathsf{E}} \to \mathcal{J} \\
\mathsf{step}/¶_{\mathsf{E}} &: \{X, e, c\}\ \bigcirc(\mathsf{step}^c(e) = e) \\
\bigcirc^+ &: \mathsf{tp}^+ \to \mathsf{tp}^+ \\
\_ &: \{A\}\ \mathsf{tm}^+(\bigcirc^+ A) \cong \bigcirc(\mathsf{tm}^+(A))
\end{aligned}
\qquad
\begin{aligned}
\mathsf{eq} &: (A : \mathsf{tp}^+) \to \mathsf{tm}^+(A) \to \mathsf{tm}^+(A) \to \mathsf{tp}^+ \\
\mathsf{self} &: \{A\}\ (a, b : \mathsf{tm}^+(A)) \to \\
&\quad a =_{\mathsf{tm}^+(A)} b \to \mathsf{tm}^+(\mathsf{eq}_A(a, b)) \\
\mathsf{ref} &: \{A\}\ (a, b : \mathsf{tm}^+(A)) \to \\
&\quad \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{eq}_A(a, b))) \to a =_{\mathsf{tm}^+(A)} b \\
\mathsf{uni} &: \{A, a, b\}\ (p, q : \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{eq}_A(a, b)))) \to \\
&\quad \bigcirc(p = q)
\end{aligned}
$$

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

Figure 2.1: Equational presentation of **calf** as a signature $\Sigma_{\mathbf{calf}}$ in the logical framework. Here the type isOrderedMonoid encodes all the structure of an ordered monoid and Σ denotes the framework-level dependent sum. We write $(\alpha, \beta) : A \cong B$ when α and β are the forward map and backward map of an isomorphism $A \cong B$.

Return
(Calf.Types.Bounded.bound/ret)

$$\frac{}{\mathsf{IsBounded}\,(A; \mathsf{ret}(a); 0)}$$

Step
(Calf.Types.Bounded.bound/step)

$$\frac{\mathsf{IsBounded}\,(A; e; c)}{\mathsf{IsBounded}\,\left(A; \mathsf{step}^{d}(e); d + c\right)}$$

Bind
(Calf.Types.Bounded.bound/bind)

$$\frac{\mathsf{IsBounded}\,(A; e; c) \qquad \forall a : A.\, \mathsf{IsBounded}\,(B; f(a); d(a))}{\mathsf{IsBounded}\,(B; \mathsf{bind}(e; f); \mathsf{bind}(e; \lambda a.\, c + d(a)))}$$

Relax
(Calf.Types.Bounded.bound/relax)

$$\frac{\mathsf{IsBounded}\,(A; e; c) \qquad c \leq c'}{\mathsf{IsBounded}\,(A; e; c')}$$

Figure 2.2: Cost refinement lemmas in **calf** displayed in inference rule style.

(2.2.6∗2) In Section 2.2.10, we prove that "extensional cost bounds" $\bigcirc(c \leq c')$ are equivalent to ordinary cost bounds $c \leq c'$ for a large class of cost monoids in the intended model of **calf**. The purpose of such a theorem is to interpret the meaning of cost bounds derived in **calf**: when we have a proof of the refinement $\mathsf{IsBounded}\,(A; e; c)$, we know that $e = \mathsf{step}^{c'}(\mathsf{ret}(a))$ for some $c' : \mathbb{C}$ and $a : \mathsf{tm}^{+}(A)$ such that $\bigcirc(c' \leq c)$ holds. For this bound to be meaningful one needs to be able to conclude from the extensional inequality $\bigcirc(c' \leq c)$ that the expected ordinary inequality $c' \leq c$ also holds.

(2.2.6∗3) *Cost refinement lemmas.* **calf** admits many expected principles for reasoning about the isBounded refinement, and I present three representative cases in Fig. 2.2.

2.2.7. Encoding general recursive algorithms in calf.

(2.2.7∗1) The last major component of **calf** is the representation of general recursive programs, an *a priori* nontrivial obstacle. Here tension mounts in two opposite directions: on the one hand, totality and termination is essential in the logical interpretation of types as mathematical propositions in type theory; on the other hand, algorithms are generally defined using unbounded recursion for efficiency reasons. Therefore a type-theoretic framework for algorithm analysis needs to be able to encode general recursive algorithms in a way that preserves the cost intention of the original program.

(2.2.7∗2) A well-known and versatile solution to the encoding of general recursive functions in total type theory is the Bove–Capretta method [BC05]. Any general recursive program gives rise to an *accessibility predicate* that tracks the pattern of recursive calls; this accessibility predicate can be glued onto the original program as a termination metric, and the final (total) function is defined by proving that every input is accessible.

(2.2.7∗3) As I have alluded to in the discussion of **catt** **(2.1.4∗4)**, working in a cost-aware setting automatically equips the user with a way to encode general recursive programs at no additional cost. The idea is to parameterize a given program in a *clock*, induced by the cost recurrence, which can then serve as a termination metric that frees the program to make whatever recursive calls are required. As observed in Niu and Harper [NH20], the cost-aware setting evinces a synergetic relationship between cost analysis itself and programming with general recursion that is further amplified in **calf**: cost structure enables one to effectively encode general recursion, and general recursion gives rise to programs with interesting cost structure.

(2.2.7∗4) *Clocked programming.* Our experience with **calf** has shaped the intuition outlined in **(2.2.7∗3)** into a general recipe for defining and analyzing general recursive algorithms. Suppose one is given an algorithm $f : A \rightharpoonup B$ along with its *cost model.* Notice that the symbol $\rightharpoonup$ indicates that this is a *partial function.* Thus one should think of f as an informal description of an algorithm external to **calf**.

1. Define a *clocked* version of the algorithm $f_{◔} : \mathbb{N} \to A \to B$ in which a clock variable of type $\mathbb{N}$ represents the available "fuel" that is burned by making recursive calls; when the clock is nonzero, $f_{◔}$ follows the recursion pattern exhibited by f by decrementing the clock, and when the clock is zero, $f_{◔}$ terminates by returning a default value or raising an exception. step's should be placed in $f_{◔}$ in accordance with the given cost model.

2. Define the the associated *cost recurrence* for the clocked algorithm $f/\$_{◔} : \mathbb{N} \to A \to B$.

3. Define the *recursion depth* $f_{\mathsf{depth}} : A \to \mathbb{N}$ that bounds the number of recursive calls made by f on a given input $a : A$.

4. Obtain the *complete programs* by instantiating the clocked programs with the recursion depth: $f(a) = f_{◔}(f_{\mathsf{depth}}(a))(a)$ and $f/\$(a) = f/\$_{◔}(f_{\mathsf{depth}}(a))(a)$.

5. Prove that the resulting algorithm is bounded by the cost recurrence $f/\$$. This process is mostly mechanical: one repeatedly applies the lemmas in **(2.2.6∗3)** to break down $\mathsf{isBounded}$ goals.

6. Characterize the recurrence $f/\$$ by (*e.g.*) computing a closed-form solution. Usually this step represents the bulk of the work in pen-and-paper algorithm analysis.

(2.2.7∗5) *Relationship to the normal form theorem.* One of the most well-known results of computability theory Kleene [Kle43] is that any partial computable function of type $\mathbb{N} \to \mathbb{N}$ may be defined using one minimization operation; in other words, one "while loop" is sufficient to compute any partial function. We observe that the encoding of general recursive programs in **calf** shares a similar flavor in the sense that the call-graph of an encoded algorithm may be seen as counting down a single outer "for loop" whose bound is determined by the cost bound of the algorithm.

2.2.8. Parallelism in calf.

(2.2.8∗1) Parallelism arises naturally in the setting of **calf** via an equational presentation of the profiling semantics of Blelloch and Greiner [BG95]. Here we present a version adapted from Harper [Har18] in which it is observed that the source of parallelism can be isolated to the treatment of *pairs* of computations: a parallel computation of $A \times B$ is furnished by a new computation form & that conjoins two independent computations of A and B:

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

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

(2.2.8∗2) *Cost structure of parallelism.* Blelloch and Greiner [BG95] characterize the complexity of a program in terms of two measures: *work*, which represents its sequential cost, and *span*, which represents its parallel cost. In **calf** this structure is recorded by the *parallel cost monoid* $\mathbb{C} := (\mathbb{N}^2, \oplus, (0,0), \leq_{\mathbb{N}^2})$ in which $\oplus$ and $\leq_{\mathbb{N}^2}$ are component-wise extensions of addition and $\leq$. Parallel cost composition is then implemented by the operation $(w_1, s_1) \otimes (w_2, s_2) := (w_1 + w_2, \max(s_1, s_2))$ that takes the sum of the works and max of the spans. This provides the required structure to assemble the cost of a completed parallel pair:

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

2.2.9. Case studies.

(2.2.9∗1) One lesson I learned from **catt** is that mechanization and cost analysis is a one-two punch: many of the benefits of a formal framework cannot be grasped on paper alone. In this sense this **calf** represents a major advancement over **catt**. The examples we have studied in **calf** so far are not state-of-the-art but include heavy hitters usually found in an introductory algorithms textbook: Euclid's algorithm, sequential and parallel insertion and merge sort, and amortized analysis of batched queues. For all of the algorithms except parallel merge sort we have verified the best known asymptotic bound, a feat that relies crucially on the ability to express extensional/behavioral specifications and use ordinary mathematical reasoning in **calf**. This small collection of case studies suggests an auspicious beginning to a growing library of formally verified algorithms in **calf**, which would be the first steps towards the development of large-scale cost-aware programs. In the following I will try to convey the experience of using **calf** for program verification.

(2.2.9∗2) *Agda encoding of **calf**.* We define **calf** in Agda by postulating the constants in the signature $\Sigma_{\mathbf{calf}}$ (see Fig. 2.1) and animating the associated equations with the recently added rewriting facilities [CTW21]. For instance, the basic judgmental structure of **calf** may be specified by the following Agda postulates:

$$
\begin{array}{lll}
\textbf{postulate} & \textbf{postulate} & \textbf{postulate} \\
\quad \mathsf{mode} : \mathbf{Set} & \quad \mathsf{tp} : \mathsf{mode} \to \mathbf{Set} & \quad \mathsf{F} : \mathsf{tp}\ \mathsf{pos} \to \mathsf{tp}\ \mathsf{neg} \\
\quad \mathsf{pos} : \mathsf{mode} & \quad \mathsf{tm}^{+} : \mathsf{tp}\ \mathsf{pos} \to \mathbf{Set} & \quad \mathsf{U} : \mathsf{tp}\ \mathsf{neg} \to \mathsf{tp}\ \mathsf{pos} \\
\quad \mathsf{neg} : \mathsf{mode} & &
\end{array}
$$

In this encoding of **calf** we define computations as $\mathsf{tm}^{\ominus}(X) = \mathsf{tm}^{+}(\mathsf{U}(X))$, leading to a more streamlined version of CBPV in which thunk and force are identities. Observe that the Agda implementation of **calf** constitutes an algebra **(2.2.10∗3)** $\mathcal{A}_{\mathrm{Agda}}$ in which **Set** plays the role of the universe of judgments **Jdg**.

(2.2.9∗3) The equational theory of **calf** is encoded via rewriting; for instance we may implement the inversion principle for bind as follows:

$$
\begin{array}{l}
\textbf{postulate} \\
\quad \mathsf{bind/ret} : \{A, X\}\ \{v : \mathsf{tm}^{+}(A)\}\ \{f : (x : \mathsf{tm}^{+}(A)) \to \mathsf{tm}^{\ominus}(X)\} \\
\qquad\qquad \mathsf{bind}(\mathsf{ret}(v); f) \equiv f(v) \\
\qquad\qquad \text{\{-\# REWRITE bind/ret \#-\}}
\end{array}
$$

Here the REWRITE pragma tells Agda to treat bind/ret as a rewrite rule that replaces the expression to the left of $\equiv$ with the expression to the right.

(2.2.9∗4) *Cost bounds.* Following the description of **(2.2.6∗1)**, the data associated with cost bounds is naturally captured by a record type in the Agda encoding of **calf**:

$$\begin{aligned}
&\textbf{record}\ \mathsf{IsBounded}(A : \mathsf{tp}^{+})(e : \mathsf{tm}^{\ominus}(\mathsf{F}(A)))(c : \mathbb{C}) : \textbf{Set where}\\
&\quad \mathsf{result} : \mathsf{tm}^{+}(A)\\
&\quad c' : \mathbb{C}\\
&\quad \mathsf{hyp/bounded} : \bigcirc(c' \leq c)\\
&\quad \mathsf{hyp/eq} : e \equiv \mathsf{step}^{c'}(\mathsf{ret}(\mathsf{result}))
\end{aligned}$$

(2.2.9∗5) *Cost models.* As indicated in the recipe from **(2.2.7∗4)**, the analysis of every algorithm begins with the definition of the cost model. In Euclid's algorithm the cost model is the number modulus operations. In **calf** is this specified by an instrumented version of the modulus operation that incurs unit cost:

$$\begin{aligned}
\mathit{mod}_{\mathsf{inst}} &: \mathsf{tm}^{+}(\mathsf{nat}) \to \mathsf{tm}^{+}(\mathsf{nat}) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{nat}))\\
\mathit{mod}_{\mathsf{inst}}(x, y) &= \mathsf{step}^{1}(\mathsf{ret}(\mathit{mod}(x, y)))
\end{aligned}$$

On the other hand, in the analysis of sorting algorithms, it is customary to consider the comparison cost model in which the only operation that incurs cost is the comparison operation. In **calf** we may parameterize the analyses of sorting algorithms by the following *comparable* type:

$$\begin{aligned}
&\textbf{record}\ \mathsf{Comparable} : \textbf{Set}_1\ \textbf{where}\\
&\quad A : \mathsf{tp}^{+}\\
&\quad \leq\ : \mathsf{tm}^{+}(A) \to \mathsf{tm}^{+}(A) \to \textbf{Set}\\
&\quad \leq_{\mathsf{dec}} : \mathsf{tm}^{+}(A) \to \mathsf{tm}^{+}(A) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{bool}))\\
&\quad \leq_{\mathsf{dec}} / \leq\ : \{x, y, b\} \to \bigcirc((x \leq_{\mathsf{dec}} y) \equiv \mathsf{ret}(b) \to \mathsf{Reflects}\ (x \leq y)\ b)\\
&\quad \leq /\mathsf{ord} : \mathsf{isTotalOder}\ \leq\\
&\quad \leq_{\mathsf{dec}} /\mathsf{cost} : (x, y : \mathsf{tm}^{+}(A)) \to \mathsf{IsBounded}\ \mathsf{bool}\ (x \leq_{\mathsf{dec}} y)\ 1
\end{aligned}$$

In other words a comparable type is a type A equipped with a total ordering relation $\leq$. To program with comparable types we also need the ordering to be decidable, which is encoded above as $\leq_{\mathsf{dec}}$. Because the comparison cost model dictates that the comparison operation is unit cost, we require a field $\leq_{\mathsf{dec}}$ /cost to record this fact using the $\mathsf{IsBounded}$ type defined in **(2.2.9∗4)**. Lastly, the field $\leq_{\mathsf{dec}}$ / $\leq$ indicates that $\leq_{\mathsf{dec}}$ is a decision procedure for $\leq$, *i.e.* the $x \leq_{\mathsf{dec}} y$ computes the value $\mathsf{tt} : \mathsf{bool}$ if and only if $x \leq y$ holds. Here we observe another use of the extensional modality $\bigcirc$: because the decision procedure $\leq_{\mathsf{dec}}$ is a cost-aware computation, we descend to the extensional phase to state its behavioral specification.

(2.2.9∗6) *Cost-aware programming.* With the cost model in place, cost-aware programming in **calf** is very similar to ordinary programming. For instance, we may implement insertion sort by following the standard textbook definition adapted to a CBPV setting:

$$
\begin{aligned}
\mathit{insert} &: \mathsf{tm}^{+}(A) \to \mathsf{tm}^{+}(\mathsf{list}(A)) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{list}(A))) \\
\mathit{insert}\, x\ [\,] &= \mathsf{ret}([x]) \\
\mathit{insert}\, x\ (y :: ys) &= \\
&\quad \mathsf{bind}\ (x \leq_{\mathsf{dec}} y)\ (\lambda b. \\
&\quad \mathsf{case}\ b\ \mathsf{of} \\
&\qquad \mathsf{tt} \to \mathsf{bind}\ (\mathit{insert}\, x\ ys)\ (\lambda r.\, \mathsf{ret}(y :: r)) \\
&\qquad \mathsf{ff} \to \mathsf{ret}(x :: (y :: ys)))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{insertionSort} &: \mathsf{tm}^{+}(\mathsf{list}(A)) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{list}(A))) \\
\mathit{insertionSort}\, [\,] &= [\,] \\
\mathit{insertionSort}\, (x :: xs) &= \mathsf{bind}\ (\mathit{insertionSort}\, xs)(\lambda xs'.\, \mathit{insert}\, x\ xs')
\end{aligned}
$$

In the above we are assuming A is a comparable type in the sense of **(2.2.9∗5)**.

(2.2.9∗7) *Clocked programming.* In general, we will not be able to define verbatim the standard versions of algorithms in **calf**. As I discussed in Section 2.2.7, general recursive algorithms is encoded in **calf** by means of clocked programming. In the case of Euclid's algorithm, we define the following clocked program:

$$
\begin{aligned}
\mathit{gcd}_{\text{◔}} &: \mathsf{tm}^{+}(\mathsf{nat}) \to \mathsf{tm}^{+}(\mathsf{nat}^2) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{nat})) \\
\mathit{gcd}_{\text{◔}}\ \mathsf{zero}\ x\ y &= \mathsf{ret}(x) \\
\mathit{gcd}_{\text{◔}}\ (\mathsf{suc}(k))\ x\ \mathsf{zero} &= \mathsf{ret}(x) \\
\mathit{gcd}_{\text{◔}}\ (\mathsf{suc}(k))\ x\ (\mathsf{suc}(y)) &= \mathsf{bind}\ (\mathit{mod}_{\mathsf{inst}}(x, \mathsf{suc}(y)))\ (\lambda r.\, \mathit{gcd}_{\text{◔}}(k)(\mathsf{suc}(y), r))
\end{aligned}
$$

Recall that $\mathit{mod}_{\mathsf{inst}}$ is the instrumented modulus that encodes the cost model for Euclid's algorithm. Here the first argument to $\mathit{gcd}_{\text{◔}}$ is a clock parameter that ticks down at each recursive call of Euclid's algorithm; when the clock parameter is zero $gcd_{◔}$ simply returns the first real argument. In other words $gcd_{◔}\ k\ x\ y$ is the k-approximation of $\gcd(x, y)$ in which Euclid's algorithm is only allowed to make k recursive calls.

(2.2.9∗8) *Instantiating the clock.* Does the clocked program $gcd_{◔}$ satisfies the behavioral specification of Euclid's algorithm, *i.e.* does $gcd_{◔}$ compute the gcd? We already observed that $gcd_{◔}\ k\ x\ y$ only computes k-approximations of $\gcd(x, y)$. However, if the approximation level k (*i.e.* number of recursive calls) is sufficient, then it should be the case that $gcd_{◔}\ k\ x\ y$ actually computes $\gcd(x, y)$. First, we define a function to compute the necessary approximation level (*i.e.* depth of the recursion tree) for any input to Euclid's algorithm:

$$gcd_{\mathsf{depth}} : \mathsf{tm}^{+}(\mathsf{nat}^{2}) \to \mathsf{tm}^{+}(\mathsf{nat})$$

$$gcd_{\mathsf{depth}}\ x\ y = \begin{cases} \mathsf{zero} & \text{if y} = \mathsf{zero} \\ \mathsf{suc}(gcd_{\mathsf{depth}}\ y\ (mod\,(x, y))) & \text{o.w.} \end{cases}$$

Note that gcd_{depth} is a specification of a function by cases: because we do not need to track the cost of computing the recursion depth, gcd_{depth} may be defined however convenient.[4]

(2.2.9∗9) *Extensional modality.* We now instantiate the clocked algorithm $gcd_{◔}$ by the recursion depth gcd_{depth}: define $gcd\ x\ y := gcd_{◔}\ (gcd_{\mathsf{depth}}\ x\ y)\ x\ y$. We may prove that gcd computes the gcd, a behavioral specification that is naturally expressed as the following equations in the extensional phase:

$$\bigcirc(gcd\,x\ \mathsf{zero} = \mathsf{ret}(x)) \tag{1}$$

$$\bigcirc(gcd\,x\ (\mathsf{suc}(y)) = gcd\ (\mathsf{suc}(y))\ (mod\,x\ \mathsf{suc}(y))) \tag{2}$$

As we mentioned previously **(2.2.6∗1)**, the extensional modality also appears when reasoning about cost bounds themselves **(2.2.9∗13)**.

(2.2.9∗10) *Cost recurrences.* The method of recurrence relations in traditional presentations of algorithm analysis is divided into the two expected stages in **calf**: we extract a cost recurrence \$ from the algorithm and compute a closed-form formula ϕ for the cost recurrence \$. Each of these steps has an associated proof obligation: we have to show that \$ is indeed a cost bound for the algorithm and that ϕ is an upper bound for \$. Recall from **(2.2.7∗4)** that a cost recurrence is a function that assigns a cost to each input of the algorithm *paired* with a given clock. In the case

[4]In Agda we define gcd_{depth} using well-founded induction on the last argument.

of Euclid's algorithm we have the following cost recurrence associated to $\mathit{gcd}_{◔}$:

$$\begin{aligned}
\mathit{gcd}/\$_{◔} &: \mathsf{tm}^{+}(\mathsf{nat}) \to \mathsf{tm}^{+}(\mathsf{nat}^2) \to \mathsf{tm}^{+}(\mathsf{nat}) \\
\mathit{gcd}/\$_{◔}\ \mathsf{zero}\ x\ y &= \mathsf{zero} \\
\mathit{gcd}/\$_{◔}\ (\mathsf{suc}(k))\ x\ y &= \begin{cases} \mathsf{zero} & \text{if y} = \mathsf{zero} \\ \mathsf{suc}(\mathit{gcd}/\$_{◔}\ k\ y\ (\mathit{mod}(x,y))) & \text{o.w.} \end{cases}
\end{aligned}$$

Similar to the case for the recursion depth **(2.2.9∗8)**, we do not track the cost computing the cost recurrence.

(2.2.9∗11) We may prove that $\mathit{gcd}/\$_{◔}$ is a cost bound for $\mathit{gcd}_{◔}$, which is expressed by the following **calf** theorem:

$$\mathit{gcd}_{◔}/\mathsf{bound} : (k,x,y : \mathsf{tm}^{+}(\mathsf{nat})) \to \mathsf{IsBounded}\ \mathsf{nat}\ (\mathit{gcd}_{◔}\ k\ x\ y)\ (\mathit{gcd}/\$_{◔}\ k\ x\ y)$$

The proof of $\mathit{gcd}_{◔}/\mathsf{bound}$ is entirely mechanical: the user simply breaks down the overall IsBounded proof goal and fulfills the generated sub-goals using the syntax-directed cost refinement lemmas (some of which are depicted in Fig. 2.2). In fact this step is taken to be so obvious that often no proof is given in textbook presentations of algorithm analysis.

(2.2.9∗12) The last and usually most difficult step in algorithm analysis is to compute a closed-form solution or otherwise informative bound to the cost recurrence. This step is also the place where **calf** shines as a mathematical domain for reasoning about cost bounds. For instance, we may prove a very precise bound on the cost recurrence $\mathit{gcd}/\$\ x\ y := \mathit{gcd}/\$_{◔}\ (\mathit{gcd}_{\mathsf{depth}}\ x\ y)\ x\ y$ that is known to be asymptotically tight.[5] Let $\mathsf{Fib} : \mathbb{N} \to \mathbb{N}$ be the fibonacci sequence, and let $\mathsf{Fib}^{-1} : \mathbb{N} \to \mathbb{N}$ be the function characterized by the equation $\mathsf{Fib}^{-1}(x) = \max\{i \mid \mathsf{Fib}(i) \leq x\}$. We have the following **calf** theorem:

$$\mathit{gcd}/\$/\mathsf{bound} : (x,y : \mathsf{tm}^{+}(\mathsf{nat})) \to (x > y) \to \mathit{gcd}/\$\ x\ y \leq 1 + \mathsf{Fib}^{-1}(x)$$

In conjunction with **(2.2.9∗11)**, we obtain the following cost bound for *gcd*:

$$\mathit{gcd}/\mathsf{bound} : (x,y : \mathsf{tm}^{+}(\mathsf{nat})) \to (x > y) \to \mathsf{IsBounded}\ \mathsf{nat}\ (\mathit{gcd}\ x\ y)\ (1 + \mathsf{Fib}^{-1}(x))$$

(2.2.9∗13) *Extensional cost bounds.* As I mentioned in **(2.2.6∗1)**, it is sometimes necessary to access the extensional phase of **calf** when proving bounds on cost

[5]We have not verified that it is indeed the tightest bound possible. But **calf** allows the user to prove these results if desired.

recurrences. For instance, when computing the closed-form solution to the cost recurrence of *insertionSort* from **(2.2.9∗6)**, we would like to have access the theorem *insertionSort*/correct : IsSort *insertionSort* stating that *insertionSort* is a sorting algorithm, where IsSort is the following family:

$$\begin{aligned} \mathsf{IsSort} &: (\mathsf{tm}^{+}(\mathsf{list}(A)) \to \mathsf{tm}^{\ominus}(\mathsf{F}(\mathsf{list}(A)))) \to \mathbf{Set} \\ \mathsf{IsSort}\ f &= (l : \mathsf{tm}^{+}(\mathsf{list}(A))) \to \bigcirc(\Sigma l' : \mathsf{tm}^{+}(\mathsf{list}(A)).\, f\ l \equiv l' \times \mathsf{SortedOf}\ l\ l') \end{aligned}$$

Observe that the predicate IsSort is naturally lives in the extensional phase because we do not want to keep track of the cost incurred by the candidate sorting program. Consequently we relate *insertionSort*/\$ (the cost recurrence associated to *insertionSort*) to its closed-form solution in the extensional phase:

$$\mathit{insertionSort}/\$/\mathsf{bound} : (l : \mathsf{tm}^{+}(\mathsf{list}(A))) \to \bigcirc(\mathit{insertionSort}/\$\ l \leq |l|^2)$$

2.2.10. Metatheory.

(2.2.10∗1) In Niu et al. [Niu+22] we substantiated the axioms of **calf** by means of a model construction and proved the following theorems:

1. ***Nondegeneracy.*** The cost effect step is not degenerate, i.e $\nvdash \mathsf{step}^1(e) = e$ for any $e : \mathsf{F}(A)$.

2. ***Validity of cost bounds***. We have that $\vDash \bigcirc(m \leq n)$ if and only if $\vDash m \leq n$ for all $m, n : \mathbb{N}$.

The first theorem states that when the cost monoid is instantiated as $\mathbb{C} := \mathbb{N}$ the cost effect step is non-degenerate; the second theorem is a statement about the *models* of **calf** and states that in the intended model of **calf** extensional cost bounds in the sense of **(2.2.6∗1)** coincide with ordinary cost bounds.

(2.2.10∗2) *Models of* ***calf***. Unlike **catt** in which the analytic construction of the theory automatically implies logical consistency and the validity of cost-aware judgments, the validity of **calf** as a theory is substantiated by means of model construction. Here we benefit from the definition of **calf** as a signature in the logical framework: **calf** is the *free* lccc $\mathscr{C}_{\mathbf{calf}}$ over the signature $\Sigma_{\mathbf{calf}}$ presented in Fig. 2.1. Consequently one may prove metatheorems about **calf** using the universal property of freely generated categories. In the context of functorial semantics [Law63], the universal property states that one may define a model $\mathscr{C}_{\mathbf{calf}} \to \mathscr{E}$ by simply specifying the image of the constants of $\Sigma_{\mathbf{calf}}$ in $\mathscr{E}$.[6]

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

(2.2.10∗3) *Algebra for a signature in the logical framework.* The data of this specification is encapsulated by the notion of an *algebra* for a signature: let $\mathscr{E}$ be a category that has a universe $\mathcal{U}$ closed under dependent products, dependent sums, and extensional equality. Given a signature Σ in the logical framework, we can define a type $\mathbf{Alg}_{\mathcal{U}}(\Sigma)$ of $\mathcal{U}$-small algebras for Σ in $\mathscr{E}$ by interpreting **Jdg** as $\mathcal{U}$ and taking the dependent sum over all the constants declared in Σ.

(2.2.10∗4) Thus, given a category $\mathscr{E}$ with the structures described in **(2.2.10∗3)**, we can define a model of **calf** by exhibiting an algebra $\mathcal{A} : \mathbf{Alg}_{\mathcal{U}}(\Sigma_{\mathbf{calf}})$ in some universe $\mathcal{U}$ of $\mathscr{E}$. In fact we can define the intended model of **calf** in *any* given topos X with a distinguished subterminal object representing the phase separation of intension and extension.

(2.2.10∗5) Given a large enough universe level λ, one may define a $\mathcal{U}_\lambda$-small model of **calf** via a version of the standard Eilenberg–Moore interpretation of CBPV in which computation types are interpreted as algebras for a given monad. In the case of **calf** we dub this interpretation the *counting model* $\mathcal{A}$, so named because the interpretation of the computation type $\mathsf{F}(A)$ is the free algebra of a particular writer monad whose carrier classifies elements of A paired with a step count.

(2.2.10∗6) We may instantiate generic construction of the counting model $\mathcal{A}$ in the presheaf topos over the interval category $\{ 0 \to 1 \}$, which may be seen to be equivalent to the arrow category $\mathbf{Set}^{\to}$. Observe that objects in this category are families of sets $A : A_1 \to A_0$, which corresponds to the fact that a type A is a family indexed in a collection of *behaviors* with the fibers representing the *cost structure* over a given behavior. In $\mathbf{Set}^{\to}$ the extensional phase $¶_{\mathsf{E}}$ is furnished by the subterminal family $0 \to 1$, and the extensional modality takes a family $A_1 \to A_0$ to the identity $A_0 \to A_0$, trivializing the fiber (*i.e.* cost structure) over each point in A_0. We write $\mathcal{A}_{\mathbf{Set}^{\to}}$ for the instance of the counting model constructed in the arrow category $\mathbf{Set}^{\to}$.

(2.2.10∗7) The counting model of **calf** validates the non-degeneracy theorem. More precisely, we may show that $(\mathsf{step}^c(e) = e) \to ¶_{\mathsf{E}}$ for any nonzero $c : \mathbb{C}$ and $e : \mathsf{F}(A)$. In other words we know that if step is degenerate, then the extensional phase $¶_{\mathsf{E}}$ is derivable. Observing that we placed no restrictions on the proposition $¶_{\mathsf{E}}$ in the construction of the counting model, we immediately obtain the desired theorem by instantiating $¶_{\mathsf{E}}$ with the false proposition.

(2.2.10∗8) The counting model also validates the equivalence between extensional equalities and ordinary equalities whenever the cost monoid is *extensional* in the sense that $\mathbb{C} \cong \bigcirc\mathbb{C}$ and the relation $\leq$ may be characterized using Σ and equality types.

(2.2.10∗9) For instance given the cost monoid $\mathbb{C} := \mathbb{N}$, We have that $\mathcal{A}_{\mathbf{Set}^{\rightarrow}} \vDash \bigcirc(m \leq n)$ if and only if $\mathcal{A}_{\mathbf{Set}^{\rightarrow}} \vDash m \leq n$ for all $m, n : \mathbb{N}$.

(2.2.10∗10) It may be natural to ask if **(2.2.10∗9)** holds for the syntactic model of **calf**, *i.e.* is it the case that $\vdash \bigcirc(m \leq n)$ if and only if $\vdash m \leq n$. While the backwards implication is immediate from the definition of the extensional modality, the forward implication requires the fact that canonicity holds for **calf**; in Section 2.2.10 we conjecture that the techniques of synthetic Tait computability developed by Sterling and Harper [SH21] can be used in the setting of **calf** to give a succinct proof of canonicity.

CHAPTER 3

PROPOSED WORK

3.1. METATHEORY

(3.1∗1) A basic property usually considered in the context of the metatheory of type theories is *canonicity*, which roughly states that there are enough equations for (closed) computation at base type. In **calf** one may formulate this property as follows:

> For any closed free computation $e : \mathsf{F}(A)$, there exists a unique cost c and value $a : A$ such that $e = \mathsf{step}^c(\mathsf{ret}(a))$.

Besides being of independent interest for a type theory, canonicity is also necessary for proving the metatheorem characterizing extensional cost bounds **(2.2.10∗10)**.

(3.1∗2) As usual canonicity may be proved by using a *logical relations* argument. Following the spirit of the *objective metatheory* of type theories propounded in Sterling [Ste21], I propose to proof canonicity for **calf** using the tools of *synthetic Tait computability* (STC) developed in *op. cit.* At a high level, STC provides a type-theoretic interface to construct and manipulate the syntactic and semantic structures arising in a traditional logical relations proof. It is argued by *op. cit.* (and to me, convincingly) that by embracing this synthetic perspective one may develop more succinct and modular proofs of metatheorems of type theory when compared to classic techniques. Indeed, STC has been deployed to prove (among other things [SH21; Gra21]) the last open syntactic metatheorem of cubical type theory [SA21].

(3.1∗3) In STC, every type may be seen as a computability structures that decomposes into a syntactic part and a semantic part, thus providing a language for *synthetic* manipulations of the data of logical relations. To be concrete, suppose that we would like to establish metatheorems about a theory $\mathbb{T}$ using STC. In this case the language of STC is an extensional type theory extended with the following:

1. A distinguished proof-irrelevant proposition $\mathsf{syn} : \Omega$ that mediates the phase distinction between the syntactic and semantic. We write $\bigcirc_{\mathsf{syn}}, \bullet_{\mathsf{syn}}$ for the associated open and closed modality.

2. A hierarchy of strict cumulative universes $\mathcal{U}_\alpha$ satisfying the realignment axiom [Ste21].

3. An algebra $\mathcal{A}_{\mathsf{syn}}$ the given theory $\mathbb{T}$ in which the judgments are interpreted in a syntactic open-modal universe, *i.e.* an element of $\mathbf{Alg}_{\bigcirc_{\mathsf{syn}}\mathcal{U}_\alpha}(\mathbb{T})$ for some universe $\mathcal{U}_\alpha$.

Note that the syntactic/semantic phase distinction of STC is formally analogous to the extension/intension phase distinction of **calf**. The syntactic open syn generates for any universe an open subuniverse of pure syntactic types, and the algebra $\mathcal{A}_{\mathsf{syn}}$ may be seen as an embedding of the target theory $\mathbb{T}$. To carry out a computability proof such as canonicity, we define an algebra $\mathcal{A}$ of $\mathbb{T}$ such that it is equal to $\mathcal{A}_{\mathsf{syn}}$ when we have a proof of syn. This aligment/restriction property corresponds to the fact that the computability data of *e.g.* booleans should be about the actual booleans of $\mathbb{T}$.

(3.1∗4) The benefit of STC is that one can define the computability algebra using simple type-theoretic constructions by exploiting the properties of the syntactic/open and semantic/closed modalities. Given such an algebra, the actual metatheorem may be extracted by instantiating the STC theory at a concrete topos. For canonicity, the STC topos is given by the Artin gluing of the global sections functor $\mathrm{Pr}(\mathscr{C}_{\mathbb{T}}) \to \mathbf{Set}$, where $\mathscr{C}_{\mathbb{T}}$ is the category of contexts of the theory $\mathbb{T}$.

(3.1∗5) I plan to use STC to prove canonicity of **calf** by adapting the construction of the canonicity model for MLTT given in Sterling [Ste21]. An anticipated difficulty is how to handle the value-computation dichotomy present in **calf**. While the computability data attached to a value type A is *any* type that restricts to syntactic values of A, the data attached to a computation type X appears to be an *algebra* for the cost monad that restricts to the syntactic algebra associated to X (that is, the algebra on $\mathsf{Alg}_{\mathsf{syn}}.\mathsf{tm}^{\ominus}(X)$ whose structure is given by the syntax of **calf**). In order to make this interpretation work, we need to verify that one may also realign along partial isomorphisms of algebras.

(3.1∗6) I have not addressed the analogous property for *open* computations of **calf**, *i.e.* normalization, which would be necessary to show that it is possible to implement proof assistants based on **calf**. The reason is two-fold. First, normalization is a much more difficult metatheorem (although certainly within reach in view of Sterling [Ste21]). Second, it is unclear what the normal forms of partial computations should be when we extend **calf** to study *denotational semantics* in Section 3.2. In any case, canonicity represents a first step in the metatheory of **calf** and should provide some insight on how to go about studying normalization.

(3.1∗7) Note that the lack of a normalization result does not prevent us from using **calf** in practice. As we outline in Section 2.2.9, it is straightforward to encode a version of **calf** in modern proof assistants, a strategy that also allows one to use the facilities and mathematical results of the host language.

3.2. SYNTHETIC COST-AWARE DENOTATIONAL SEMANTICS

The latter half of this proposal is composed of a series of experiments that positions (extensions of) **calf** as a logical framework/metalanguage for doing *synthetic* denotational semantics. In the following, we outline the motivations of this work and present the blueprint for three successively more involved case studies that furnish evidence in support of the aforementioned hypothesis.

3.2.1. Cost-aware denotational semantics.

(3.2.1∗1) In Section 2.2 we showed how to use **calf** for cost-aware programming and verification, which substantiates the view of **calf** as a cost-aware programming and specification language. On the other hand, because **calf** is also a dependent type theory, one is prompted to look for mathematics that may be naturally expressed in **calf**. Given that we have developed **calf** to express cost-aware program specifications, one natural candidate is the theory of *denotational semantics* of programming languages. Therefore, in this sense one may also view **calf** as a *metalanguage* for studying the global, *cost-sensitive* properties of programming languages.

(3.2.1∗2) In addition to be of independent interest, cost-aware denotational semantics also provides a criterion for the adequacy of cost models in view of the discussion in Section 1.1.5. We briefly sketch the idea. Suppose that we have specified in **calf** a programming language $\mathbf{P}$ along with its operational semantics $\Downarrow$. We write e for terms of $\mathbf{P}$. Moreover, assume that we also have in hand a denotational semantics $[\![-]\!]$ for $\mathbf{P}$. Consider the following property:

> Given an equation $[\![\mathsf{e}]\!] = \mathsf{ret}([\![\mathsf{a}]\!])$ at base type, we have that $\mathsf{e} \Downarrow \mathsf{a}$ operationally as well.

A denotational semantics $[\![-]\!]$ with the property above is said to be *computationally adequate* with respect to $\Downarrow$.

In order to express *effective* adequacy, we will generalize this notion by connecting operational and denotational *cost* semantics, which we refer to as *cost-aware computational adequacy.* We defer the specifics of the definition to Section 3.2.3; however the application of this property is straightforward: one may rephrase adequacy of a cost-instrumentated program e as the existence of a program e of $\mathbf{P}$ such that $[\![\mathsf{e}]\!] = e$, since by cost-aware adequacy any cost bound derived on e is guaranteed to coincide with the operational cost bound on e.

3.2.2. Synthetic mathematics.

(3.2.2∗1) We have already discussed the distinction between analytic and synthetic mathematics in Section 1.3 in the context of *theories for cost analysis.* In the present discussion we propose to extend the synthetic perspective to the theory of denotational semantics.

(3.2.2∗2) While the strength of synthetic theory for cost analysis of Sections 2.1 and 2.2 surfaced somewhat indirectly in terms of practicality and usability of the resulting theory, the benefits of synthetic denotational semantics may be grasped more directly. First, by isolating properties essential to the study of denotational semantics, one may obtain more general theorems that apply to different concrete models. Secondly, a synthetic theory is often more user-friendly. For instance, for the purposes of program verification, often times it is not necessary for the user of a framework to have a deep understanding of domain theory — what is important is that domain theory provides an *interface* for constructions and reasoning principles relevant to programming.

(3.2.2∗3) The point of synthetic denotational semantics then is to provide a collection of principles for reasoning about denotational semantics similar to that of synthetic domain theory. In my view, while synthetic domain theory provides an axiomatization of structures necessary in a topos to carry out classic domain theoretic constructions, synthetic denotational semantics can be thought of as an interface built on top of constructions of SDT that can be used to both develop new proofs and study classical proofs. This hierarchical relationship is reflected in the proposed extension of **calf** in Section 3.6 — one constructs and studies denotational models using the language of synthetic denotational semantics, the axioms of which are then validated by constructions in topoi equipped with an SDT theory.

(3.2.2∗4) Because **calf** is a synthetic theory for cost-aware constructions, when one defines a (inherently) cost-aware denotational semantics in **calf** one also automatically defines an *extensional* denotational semantics. Consequently, one may easily restrict cost-aware theorems about denotational semantics to ordinary theorems about the underlying extensional semantics. In Section 3.2.3 we show how this fact may be used to immediately extract the classic Plotkin-type computational adequacy theorem from the generalized cost-aware version.

(3.2.2∗5) This synthetic treatment of intension and extension represents an improvement compared to prior work on denotational semantics in *guarded type theory* [PMB15; MP16]. Although the denotational semantics of *op. cit.* is also intensional/cost-aware due to the use of guarded recursion, the corresponding theory of *extensional* denotational semantics does *not* follow immediately as a corollary. In fact the extensional denotational semantics requires a separate construction involving the guarded version of the delay monad and the associated weak bisimilarity relation.

The work required in *op. cit.* did not disappear; the advantage of a synthetic language of cost-aware constructions is the isolation of the interaction of intension and extension and the associated principles, which may be verified once and for all in the model and deployed more generally than the ad-hoc constructions of *op. cit.*

3.2.3. Cost-aware computational adequacy.

(3.2.3∗1) Before going into detail about the specific case studies, we first outline the ideas behind cost-aware computational adequacy. Assume we are given a cost semantics for a programming language **P**. Recall from **(1.1.1∗3)** that the cost semantics a ternary relation on closed programs, a cost monoid $\mathbb{C}$, and the set of terminal programs, and we write $\mathsf{e} \Downarrow^c \mathsf{v}$ when e evaluates to a value v with cost c. Let $[\![-]\!]$ be a denotational semantics of **P**. Consider the following attempt to generalize extensional computational adequacy:

Given an equation $[\![\mathsf{e}]\!] = \mathsf{step}^c(\mathsf{ret}([\![\mathsf{a}]\!]))$ at base type, we have that $\mathsf{e} \Downarrow^c \mathsf{a}$.

This statement does not generalize extensional adequacy because it is false in the extensional phase! Suppose $u : \P_{\mathsf{E}}$ is a proof of the extensional phase. If

$[\![\mathsf{e}]\!] = \mathsf{step}^c(\mathsf{ret}([\![\mathsf{a}]\!]))$, then $[\![\mathsf{e}]\!] = \mathsf{step}^{c'}(\mathsf{ret}([\![\mathsf{a}]\!]))$ for any $c' : \mathbb{C}$, and so by the conjectured adequacy theorem we would have $\mathsf{e} \Downarrow^c \mathsf{a}$ and $\mathsf{e} \Downarrow^{c'} \mathsf{a}$. But this is a contradiction because the cost semantics is deterministic.

(3.2.3∗2) *Phase-separated cost semantics.* The problem with the attempted generalization may be resolved by using the intensional modality **(2.2.4∗3)** to seal away cost information in the operational cost semantics. We refer to the relation as the *phase-separated evaluation relation.* Recall that the operational cost semantics may be equivalently formulated using the small-step operational semantics $(\mapsto, \mathsf{val})$. We begin by defining a version of the reflexive-transitive closure of $\mapsto$ with a cost component that is sealed by the intensional modality:

$$\textsc{Refl}\ \frac{}{e \mapsto^{\eta_\bullet 0}_{\P_{\mathsf{E}}} e} \qquad \textsc{Trans}\ \frac{e \mapsto e_1 \qquad e_1 \mapsto^{c}_{\P_{\mathsf{E}}} e_2}{e \mapsto^{c(\bullet+)\eta_\bullet 1}_{\P_{\mathsf{E}}} e_2}$$

In the above $\bullet+$ is the functorial action of $\bullet$ on $+$, and $\eta_\bullet$ is the unit of the monad. Cruically, this relation becomes equivalent to the ordinary reflexive transitive closure under the extensional phase:

Prop 3.2.3∗2.1. *Given $u : \P_{\mathsf{E}}$, we have that $e \mapsto^{\eta_\bullet c}_{\P_{\mathsf{E}}} v$ if and only if $e \mapsto^* v$ for all $c : \mathbb{N}$.*

We define the *phase-separated evaluation* as $e \Downarrow^{c}_{\P_{\mathsf{E}}} v := e \mapsto^{c}_{\P_{\mathsf{E}}} v \times v\ \mathsf{val}$, which also restricts to ordinary evaluation extensionally.

(3.2.3∗3) We now define cost-aware computational adequacy:

> Given an equation $[\![\mathsf{e}]\!] = \mathsf{step}^c(\mathsf{ret}([\![\mathsf{a}]\!]))$ at base type, we have that $\mathsf{e} \Downarrow^{\eta_\bullet c}_{\P_{\mathsf{E}}} \mathsf{a}$.

Extensional adequacy follows immediately:

Prop 3.2.3∗3.1. *Suppose that $u : \P_{\mathsf{E}}$. If $[\![\mathsf{e}]\!] = \mathsf{ret}([\![\mathsf{a}]\!])$, then $\mathsf{e} \Downarrow \mathsf{b}$.*

Proof. By cost-aware adequacy, we have $\mathsf{e} \Downarrow^{\eta_\bullet 0}_{\P_{\mathsf{E}}} \mathsf{a}$. But because phase-separated evaluation is equivalent to ordinary evaluation extensionally, we have that $\mathsf{e} \Downarrow \mathsf{a}$ as well. □

3.2.4. Synthetic denotational semantics *vs.* SDT.

(3.2.4∗1) We already observed in **(3.2.2∗3)** that synthetic denotational semantics is different from SDT. But given that a number of works already use SDT topoi to study programming languages with recursion [Sim99], recursive types [Sim04], and polymorphism [SR04], what is the purpose of the proposed distinction?

(3.2.4∗2) First, it should be noted that for *op. cit.*, a programming language with its static and dynamic semantics is specified *externally.* Then, an internal adequacy theorem is then established with respect to an internal version of the programming language defined by Godël numberings. This is used to prove the external adequacy result when the topos is *computationally 1-consistent*, *i.e.* 1-consistent for primitive recursive predicates. In **calf** there is no distinction between internal and external syntax because we view programming languages from a completely internal perspective. Consequently, there is no need to consider internal *vs.* external adequacy.

(3.2.4∗3) A more significant contribution of the proposed work stems from the fact that topoi do not directly present a convenient language for doing *cost-aware* programming/mathematics in the sense described in Section 2.2. Moreover, while both **calf** and topoi provide logics for reasoning about programs, **calf** can also be thought of as a programming language, whereas the same cannot be said for arbitrary topoi. Lastly, I believe that using **calf** to study denotational semantics also provides some pedagogical value towards the general computer science audience, who is probably more familiar with the type-theoretic language of **calf** in comparison to traditional presentations of SDT.

3.3. DENOTATIONAL SEMANTICS IN CALF

(3.3∗1) In this section we outline the case studies used to investigate **calf** as a framework for studying denotational semantics. Each case study will be centered around a programming language. The features of this language then guide extensions of the base **calf** theory with axioms necessary to define and study the associated denotational model.

(3.3∗2) The first two case studies are worked out in detail and represent encouraging evidence in support of **calf** as a framework for cost-aware denotational semantics; we have outlined initial plans for the last case study. In Section 3.4 we present **calf***, an extension of **calf** with universes and inductive types, and study the simply-typed lambda calculus (**STLC**); in Section 3.5 we extend **calf*** with unbounded iteration and use the resulting theory to study Modernized Algol (**MA**), a language with while loops and first-order store. Lastly in Section 3.6 we discuss the plan to extend the previous theories to account for general recursion and sketch a denotational semantics for PCF in the extended theory.

3.4. CASE STUDY: STLC

(3.4∗1) In this section we present **calf***, an extension of **calf** with inductive types and universes, and demonstrate the general machinery used to execute synthetic cost-aware denotational semantics in **calf*** with a very simple programming language, the simply-typed lambda calculus (**STLC**). We also construct models of **calf*** based on mild modifications of the standard model of **calf** presented in Niu et al. [Niu+22].

3.4.1. Axiomatizing calf*.

(3.4.1∗1) *Inductive types.* In order to define object languages such as the **STLC** in **calf**, we have to extend **calf** with a mechanism for defining general inductive types. By well-known results of type theory, inductive types may be encoded in an extensional type theory such as **calf** via W-types. For presentation purposes we will continue to write inductive definitions using traditional notations, but it should be noted that unless stated otherwise they are all definitions internal to **calf** and may be unraveled to a W-type.

(3.4.1∗2) *Universes.* Because the type-level component of a denotational semantics is a recursive function assigning object language types to **calf** *types*, we also need to axiomatize universes in **calf**. Following prior work on dependent call-by-push-value, we do so by introducing a pair of value and computation universes:

$$\begin{aligned}
\mathsf{Univ}^{+} &: \mathsf{tp}^{+}\\
\mathsf{El}^{+} &: \mathsf{tm}^{+}(\mathsf{Univ}^{+}) \to \mathsf{tp}^{+}\\
\mathsf{Univ}^{\ominus} &: \mathsf{tp}^{\ominus}\\
\mathsf{El}^{\ominus} &: \mathsf{tm}^{+}(\mathsf{Univ}^{\ominus}) \to \mathsf{tp}^{\ominus}
\end{aligned}$$

We then close the value and computation universe under all connectives of **calf**. As an example, closure under Π may be specified as follows:

$$\widehat{\Pi} : (A : \mathsf{Univ}^{+}) \to (\mathsf{El}^{+}(A) \to \mathsf{Univ}^{\ominus}) \to \mathsf{Univ}^{\ominus}$$
$$\widehat{\Pi}/\mathsf{decode} : \{A, X\}\ \mathsf{El}^{\ominus}(\widehat{\Pi}(A, X)) = \Pi(\mathsf{El}^{+}(A), \lambda a.\ \mathsf{El}^{\ominus}(X(a)))$$

(3.4.1∗3) We now axiomatize properties necessary for proving that the cost-aware generalization of the standard denotational semantics of **STLC** is computationally

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash \mathsf{lam}(x.e) : A_1 \Rightarrow A_2} \qquad \frac{\Gamma \vdash e : A_1 \Rightarrow A_2 \qquad \Gamma \vdash e_1 : A_1}{\Gamma \vdash \mathsf{ap}(e, e_1) : A_2}$$

$$\frac{}{\Gamma \vdash \mathsf{tt}, \mathsf{ff} : \mathsf{bool}} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash \mathsf{if}(e, e_1, e_2) : A}$$

Figure 3.1: A version of the **STLC** in **calf***.

adequate in the sense of **(3.2.3∗3)**. For **STLC**, we only require one property — the uniqueness of cost bounds:

$$\mathsf{step/inj} : \{A, (a, a' : A)(c, c' : \mathbb{C})\}\ \mathsf{step}^{c}(\mathsf{ret}(a)) = \mathsf{step}^{c'}(\mathsf{ret}(a')) \to a = a' \times \bullet(c = c')$$

Note that because the premise of step/inj could have been derived using a proof of the extensional phase, we must seal the equation $c = c'$ by the intensional modality.

(3.4.1∗4) The axioms introduced in **(3.4.1∗1)** through **(3.4.1∗3)** constitute an extension of **calf** we refer to as **calf***. In Section 3.4.2 we show how to define a computational adequate denotational model of **STLC** in **calf***.

3.4.2. Cost-aware denotational semantics of STLC.

(3.4.2∗1) We work with a version of the **STLC** with a boolean type bool and a call-by-value operational semantics. In view of **(3.4.1∗1)**, the statics and dynamics of this language may be specified in **calf** using inductive types in the standard way. We write $\Downarrow$ for the ordinary evaluation relation and $\Downarrow_{¶_\mathsf{E}}$ for the phase-separated evaluation (see **(3.2.3∗2)**) of **STLC**. For reference we present the static semantics of **STLC** in Fig. 3.1.

(3.4.2∗2) The denotational semantics of **STLC** in **calf*** is based on the standard decomposition of CBV in CBPV. The types are interpreted as follows:

$$\begin{aligned} [\![-]\!]_{\mathsf{Ty}} &: \mathsf{Ty} \to \mathsf{tp}^+ \\ [\![\mathsf{bool}]\!]_{\mathsf{Ty}} &= \mathsf{bool} \\ [\![A_1 \Rightarrow A_2]\!]_{\mathsf{Ty}} &= \mathsf{U}([\![A_1]\!]_{\mathsf{Ty}} \to \mathsf{F}([\![A_2]\!]_{\mathsf{Ty}})) \end{aligned}$$

In the above we write Ty for the type of **STLC** types. Terms of **STLC** are interpreted in the standard way. Note that to align the denotational cost with operational cost it is necessary to incur cost in locations where reductions occur in the operational semantics:

$$\begin{aligned}
[\![-]\!]_{\mathsf{Tm}} &: \{\Gamma, A\}\ \mathsf{Tm}(\Gamma, A) \to [\![\Gamma]\!]_{\mathsf{Con}} \to \mathsf{F}([\![A]\!]_{\mathsf{Ty}}) \\
[\![x]\!]_{\mathsf{Tm}} &= \mathsf{ret} \circ \pi_1 \\
[\![\mathsf{tt}]\!]_{\mathsf{Tm}}(-) &= \mathsf{ret}(\mathsf{tt}) \\
[\![\mathsf{ff}]\!]_{\mathsf{Tm}}(-) &= \mathsf{ret}(\mathsf{ff}) \\
[\![\mathsf{if}(e, e_1, e_2)]\!]_{\mathsf{Tm}}(\gamma) &= \mathsf{bind}([\![e]\!]_{\mathsf{Tm}}(\gamma); \lambda b.\ \mathsf{step}^1(\mathsf{if}(b, [\![e_1]\!]_{\mathsf{Tm}}(\gamma), [\![e_2]\!]_{\mathsf{Tm}}(\gamma)))) \\
[\![\mathsf{lam}(x.e)]\!]_{\mathsf{Tm}}(\gamma)(a) &= [\![e]\!]_{\mathsf{Tm}}(a, \gamma) \\
[\![\mathsf{ap}(e, e_1)]\!]_{\mathsf{Tm}}(\gamma) &= \mathsf{bind}([\![e]\!]_{\mathsf{Tm}}(\gamma); \lambda f.\ \mathsf{bind}([\![e_1]\!]_{\mathsf{Tm}}(\gamma); \lambda a.\ \mathsf{step}^1(f(a))))
\end{aligned}$$

In the above we write $\mathsf{Tm}(\Gamma, A)$ for the type of **STLC** terms of type A in the context Γ.

(3.4.2∗3) We will prove cost-aware computational adequacy by means of a logical relations argument. Guided by the natural value-computation dichotomy present in the metalanguage, we define a binary logical relation $\approx_{A:\mathsf{Ty}} \subseteq \mathsf{Pg}(A) \times [\![A]\!]_{\mathsf{Ty}}$ that relates **STLC** programs to values of semantic domain and use this to define another relation $\approx^{\Downarrow}_{A:\mathsf{Ty}} \subseteq \mathsf{Pg}(A) \times \mathsf{F}([\![A]\!]_{\mathsf{Ty}})$ that relates **STLC** programs to computations of the semantic domain.[1] Here we write $\mathsf{Pg}(A)$ for closed **STLC** terms of type A. Because the value logical relation employs the computation relation at higher types we define them mutual recursively on the structure of **STLC** types:

$$\bar{b} \approx_{\mathsf{bool}} b$$

$$\mathsf{lam}(x.\mathsf{e}_2) \approx_{A_1 \Rightarrow A_2} e \quad \text{iff} \quad \forall \mathsf{e}_1 : \mathsf{Pg}(A_1), e_1 : [\![A_1]\!]_{\mathsf{Ty}}.\ \mathsf{e}_1 \approx_{A_1} e_1 \implies [\mathsf{e}_1/x]\mathsf{e}_2 \approx^{\Downarrow}_{A_2} e(e_1)$$

$$\mathsf{e}\ R^{\Downarrow}\ e \quad \text{iff} \quad \Sigma \mathsf{v} : \mathsf{Pg}(A).\ \Sigma v : [\![A]\!]_{\mathsf{Ty}}.\ (\mathsf{e} \Downarrow^{\eta \bullet c}_{\P_{\mathsf{E}}} \mathsf{v}) \times e = \mathsf{step}^c(\mathsf{ret}(v)) \times \mathsf{v}\ R\ v$$

In the above we write $\bar{b}$ for the boolean numeral of $b : \mathsf{bool}$.

(3.4.2∗4) We may prove the fundamental theorem:

Theorem 3.4.2∗4.1 (FTLR). *Given a STLC term* $e : \mathsf{Tm}(\Gamma, A)$*, if* $\gamma \approx_\Gamma \gamma$*, then* $e[\gamma] \approx^{\Downarrow}_A [\![e]\!]_{\mathsf{Tm}}(\gamma)$.

In the above $\approx_\Gamma$ is the evident generalization of the value logical relation to to a context of **STLC** types. Cost-aware computational adequacy is an immediate corollary of Theorem 3.4.2∗4.1:

[1]Technically these relations are defined as functions into tp^+, but because the fibers of each family are subterminal we use the traditional proof-irrelevant notation.

Corollary 3.4.2∗4.1 (Computational adequacy). *Given a closed term $e : \mathsf{Pg}(\mathsf{bool})$, if $e = \mathsf{step}^c(\mathsf{ret}(b))$ for some $c : \mathbb{N}$ and $b : \mathsf{bool}$, then $e \Downarrow^{\eta\bullet c} \bar{b}$.*

Moreover, we also easily obtain the ordinary extensional adequacy theorem:

Corollary 3.4.2∗4.2 (Extensional adequacy). *Suppose that $u : ¶_{\mathsf{E}}$. Given a closed term $e : \mathsf{Pg}(\mathsf{bool})$, if $e = \mathsf{ret}(b)$ for some $b : \mathsf{bool}$, then $e \Downarrow \bar{b}$.*

3.4.3. Models of calf*.

(3.4.3∗1) Recall from Section 2.2.10 that one may define the *counting model* of **calf** in any presheaf topos with a distinguished proposition representing the extensional phase. We will define models of **calf*** based on the counting model.

(3.4.3∗2) *Modeling inductive types.* Because W types exist in any topos equipped with a natural numbers object, we may interpret W types of **calf*** in any presheaf topos.

(3.4.3∗3) *Modeling universes.* In the counting model, value types of **calf** are interpreted as plain types in the model and computations types are interpreted as algebras for the cost monad $\bullet\mathbb{C} \times -$. Therefore we simply interpret the value universe Univ^{+} as a sufficiently large universe $\mathcal{U}_\alpha$ of $\mathbf{X}$. Because the computation universe $\mathsf{Univ}^{\ominus}$ is itself a computation type we have to assign it an algebra structure. As the cost structure of types is irrelevant, we may use the trivial algebra structure associated to the projection map $\bullet\mathbb{C} \times A \to A$. The carrier of this algebra is the type of $\mathcal{U}_\alpha$-small algebras, *i.e.* algebras whose carriers are valued in $\mathcal{U}_\alpha$.

3.5. CASE STUDY: MA

(3.5∗1) The second language we study is Modernized Algol (**MA**), a variant of Algol presented in Harper [Har12]. **MA** represents a moderate increase in complexity from the **STLC**. It features while loops and first-order store and is a step closer to a language in which one may implement real-world algorithms. We follow the same outline as for the **STLC**. First, we introduce **calf**$^\omega$, an extension of **calf** with unbounded iteration. Next, we define a denotational semantics for **MA** in **calf**$^\omega$ and show that it satisfies a cost-aware adequacy theorem similar to the one presented in **(3.4.2∗4)**. Lastly, we sketch a model of **calf**$^\omega$.

3.5.1. Axiomatizing calf$^\omega$.

(3.5.1∗1) For our purposes, unbounded iteration refers to the unbounded iteration of a function $A \to \mathsf{F}(B + A)$ in which the type A represents the state of the iterative system and B is the type of the terminal state. Given such a iterative transition system $f : A \to \mathsf{F}(B + A)$, we write $\mathsf{iter}(f) : A \to \mathsf{F}(B)$ for the induced iterative computation. Iterative computations should satisfy the evident unfolding law: $\mathsf{iter}(f, a) = \mathsf{bind}(fa; [\mathsf{ret}; \mathsf{iter}(f)])$.

(3.5.1∗2) Because unbounded iteration necessarily engenders non-terminating computations, we have to arrange for these computations in **calf**$^\omega$. Although it is possible to include divergent computations in the free computations, it is simpler to introduce a type of *lifted* computations that embeds the free computations:

$$\begin{aligned} \mathsf{L} &: \mathsf{tp}^+ \to \mathsf{tp}^\ominus \\ \mathsf{lift} &: \{A\}\ \mathsf{F}(A) \to \mathsf{L}(A) \\ \mathsf{lift/inj} &: \{A\}\ \mathsf{lift}(e) = \mathsf{lift}(e') \to e = e' \end{aligned}$$

We also assume that L is a monad:

$$\begin{aligned} \mathsf{ret}_\mathsf{L} &: \{A\}\ A \to \mathsf{L}(A) \\ \mathsf{ret}_\mathsf{L} &:= \mathsf{lift} \circ \mathsf{ret} \\ \mathsf{bind}_\mathsf{L} &: \{A, B\}\ \mathsf{L}(A) \to (A \to \mathsf{L}(B)) \to \mathsf{L}(B) \end{aligned}$$

Moreover, we postulate that the both the monad structure and the cost effect commutes with lift:

$$\begin{aligned} \mathsf{lift/bind} &: \{A, B, e, f\}\ \mathsf{lift}(\mathsf{bind}(e; f)) = \mathsf{bind}_\mathsf{L}(\mathsf{lift}(e); \mathsf{lift} \circ f) \\ \mathsf{lift/step} &: \{A, e, c\}\ \mathsf{lift}(\mathsf{step}^c(e)) = \mathsf{step}^c(\mathsf{lift}(e)) \end{aligned}$$

(3.5.1∗3) *Unbounded iteration.* As mentioned in **(3.5.1∗1)**, we axiomatize that iteration is available for lifted computations:

$$\begin{aligned} \mathsf{iter} &: \{A, B\}\ (A \to \mathsf{L}(B + A)) \to A \to \mathsf{L}(B) \\ \mathsf{iter/unfold} &: \{A, B, f, a, a'\}\ \mathsf{iter}(f)(a) = \mathsf{bind}_\mathsf{L}(f(a); [\mathsf{ret}_\mathsf{L}(b); \mathsf{iter}(f)]) \end{aligned}$$

Moreover, to prove computational adequacy it is necessary to require that whenever an iterative computation is total, a finite prefix of the iteration suffices to compute the same value, which expresses a kind of compactness property:

$$\begin{aligned}
\mathsf{seq} &: \{A, B\}\ (A \to \mathsf{L}(B + A)) \to \mathbb{N} \to A \to \mathsf{L}(B + A) \\
\mathsf{seq}(f, 0)(a) &= \mathsf{ret}_\mathsf{L}(\mathsf{inr}(a)) \\
\mathsf{seq}(f, k+1)(a) &= \mathsf{bind}_\mathsf{L}(f(a); [\mathsf{ret}_\mathsf{L} \circ \mathsf{inl}; \mathsf{seq}(f, k)]) \\
\mathsf{iter/trunc} &: \{A, B, f, a, b, c\}\ \mathsf{iter}(f, a) = \mathsf{step}^c(\mathsf{ret}_\mathsf{L}(b)) \to \\
&\quad \|\Sigma k : \mathbb{N}.\ \mathsf{seq}(g, k)(a) = \mathsf{step}^c(\mathsf{ret}_\mathsf{L}(\mathsf{inl}(b)))\|
\end{aligned}$$

(3.5.1∗4) *Characterizing cost bounds.* Lastly, similar to the **calf*** we require that cost bounds are unique **(3.4.1∗3)**. Moreover, we require that cost bounds are decomposable in the following sense:

$$\begin{aligned}
\mathsf{bind}_\mathsf{L}^{-1} &: \{A, B, e, f, c, b\}\ \mathsf{bind}_\mathsf{L}(e; f) = \mathsf{step}^c(\mathsf{ret}_\mathsf{L}(b)) \to \\
&\quad \|\Sigma c_1, c_2 : \mathbb{C}.\ \Sigma a : A.\ e = \mathsf{step}^{c_1}(\mathsf{ret}_\mathsf{L}(a)) \times \\
&\quad f(a) = \mathsf{step}^{c_2}(\mathsf{ret}_\mathsf{L}(b)) \times \bullet(c = c_1 + c_2)\| \\
\mathsf{step}_\mathsf{L}^{-1} &: \{A, c, c_1, a\}\ \mathsf{step}^{c_1}(e) = \mathsf{step}^c(\mathsf{ret}_\mathsf{L}(a)) \to \\
&\quad \|\Sigma c_2 : \mathbb{C}.\ e = \mathsf{step}^{c_2}(\mathsf{ret}_\mathsf{L}(a)) \times \bullet(c = c_1 + c_2)\|
\end{aligned}$$

For technical reasons the conclusions of the axiom in **(3.5.1∗3)** and the axioms above have to be propositionally truncated [Uni13]. The decomposition of cost bounds is necessary for proving computational adequacy of the denotational semantics we define for **MA**. In contrast, because the **STLC** is a total language the analogue of these axioms were not strictly necessary in **calf***: one may state (cost-aware) computational adequacy so that uniqueness of cost bounds is sufficient for the proof of the fundamental theorem.

3.5.2. Cost-aware denotational semantics of MA.

(3.5.2∗1) In this section we define and study a denotational semantics for the programming language **MA**. The plan remains the same as the one we outlined in Section 3.4.2 for the **STLC**. For presentation purposes we do not go over every detail of the construction and simply highlight the important ideas.

(3.5.2∗2) A characteristic feature of **MA** is the distinction between *expressions*, which are meant to encode mathematical/pure computations, and *commands*, which engender effects upon execution. In Fig. 3.2 we present a fragment of the static semantics of **MA**. Observe that there are two mutually defined judgments reflecting the expression/command distinction; note that in a addition to a context, both the expression and command typing judgment is indexed by a *signature* that contains the

$$\frac{}{\Gamma, x : A \vdash_\Sigma x : A} \qquad \frac{\Gamma, x : A_1 \vdash_\Sigma e : A_2}{\Gamma \vdash_\Sigma \mathsf{lam}(x.e) : A_1 \Rightarrow A_2}$$

$$\frac{\Gamma \vdash_\Sigma e : A_1 \Rightarrow A_2 \qquad \Gamma \vdash_\Sigma e_1 : A_1}{\Gamma \vdash_\Sigma \mathsf{ap}(e, e_1) : A_2} \qquad \frac{\Gamma \vdash_\Sigma m \div A}{\Gamma \vdash_\Sigma \mathsf{cmd}(m) : \mathsf{cmd}(A)}$$

$$\frac{\Gamma \vdash_\Sigma a : A}{\Gamma \vdash_\Sigma \mathsf{ret}(a) : \mathsf{cmd}(A)} \qquad \frac{\Gamma \vdash_\Sigma e : \mathsf{cmd}(A) \qquad \Gamma, x : A \vdash_\Sigma m \div B}{\Gamma \vdash_\Sigma \mathsf{bnd}(e, x.m) \div B}$$

$$\frac{\Sigma[n] = \mathsf{bool} \qquad \Gamma \vdash_\Sigma m \div \mathsf{unit}}{\Gamma \vdash_\Sigma \mathsf{while}[n](m) \div \mathsf{unit}} \qquad \frac{\Sigma[n] = A}{\Gamma \vdash_\Sigma \mathsf{get}[n] \div A} \qquad \frac{\Sigma[n] = A \qquad \Gamma \vdash_\Sigma e : A}{\Gamma \vdash_\Sigma \mathsf{set}[n](e) \div \mathsf{bool}}$$

$$\frac{A_{\mathsf{pos}} : \mathsf{pos}(A) \qquad \Gamma \vdash_\Sigma e : A \qquad \Gamma \vdash_{\Sigma, x \sim A} m \div \mathsf{bool}}{\Gamma \vdash_\Sigma \mathsf{dcl}(e, x.m) : \mathsf{bool}}$$

Figure 3.2: Expressions and commands of **MA**.

assignables within the scope of the term. The passage from commands to expressions is mediated by the cmd type constructor: a type such as $\mathsf{cmd}(A)$ classifies reified commands computing a value of type A; the elimination form $\mathsf{bnd}(e, x.m)$ then takes a reified command e and sequences it with a command m.

(3.5.2∗3) The call-by-value operational semantics of the expressions of **MA** is defined as in the case for the **STLC**. The dynamics of commands is defined as a relation on *states* consisting of pairs of commands and *stores*, which is a finite list of values. For instance, the transition rule for the while command is defined as follows:

$$\frac{\mu[n] = \mathsf{ff}}{(\mu, \mathsf{while}[n](m)) \mapsto_{\mathsf{cmd}} (\mu, \mathsf{ret}(\star))}$$

$$\frac{\mu[n] = \mathsf{tt}}{(\mu, \mathsf{while}[n](m)) \mapsto_{\mathsf{cmd}} (\mu, \mathsf{bnd}(\mathsf{cmd}(m), -.\mathsf{while}[n](m)))}$$

For the purposes of our case study, it is critical that the store contains only data of base type, Correspondingly, one may only declare assignables of base type.[2]

[2]This may be seen from the typing rule for dcl among whose premises is the requirement that the declared assignable has base type.

(3.5.2∗4) *Kripke denotational semantics of* ***MA***. One of the main challenges of defining a denotational semantics for **MA** in comparison to the **STLC** is the prescence of the store. To express the stability of the interpretation of programs with respect to the allocation of assignables, we employ a Kripke style semantics in the interpretation of function and command types:

$$\llbracket - \rrbracket^{\mathbf{MA}}_{\mathsf{Ty}} : \mathsf{Ty}^{\mathbf{MA}} \to \mathsf{Sig} \to \mathsf{tp}^{+}$$

$$\llbracket A_1 \Rightarrow A_2 \rrbracket^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma) = \mathsf{U}((\Sigma' : \mathsf{Con}) \to \Sigma' \geq \Sigma \to \llbracket A_1 \rrbracket^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma') \to \mathsf{F}(\llbracket A_2 \rrbracket^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma')))$$

$$\llbracket \mathsf{cmd}(A) \rrbracket^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma) = \mathsf{U}((\Sigma' : \mathsf{Con}) \to \Sigma' \geq \Sigma \to \llbracket \Sigma' \rrbracket^{\mathbf{MA}}_{\mathsf{Sig}} \to \mathsf{L}(\llbracket A \rrbracket^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma') \times \llbracket \Sigma' \rrbracket^{\mathbf{MA}}_{\mathsf{Sig}}))$$

In the above we write Sig for the type of signatures and $\llbracket - \rrbracket^{\mathbf{MA}}_{\mathsf{Sig}} : \mathsf{Sig} \to \mathsf{tp}^{+}$ for the interpretation of signatures.[3] Here $\Sigma' \geq \Sigma$ is a preorder on signatures expressing the passage between worlds, where a future (*i.e.* larger) world potentially contains new allocations. Therefore a function $f : A_1 \Rightarrow A_2$ is semantically a family of functions indexed by all future worlds, and similarly a command $m : \mathsf{cmd}(A)$ is a family of store transformations that may be executed at all future worlds. Note that because commands may produce divergent computations the codomain of this transformation is a lifted type.

(3.5.2∗5) The semantics of expression and commands of **MA** follows directly from the semantics of types. As an important case, let us look at the interpretion of while loops:

$$\begin{aligned}
\llbracket \mathsf{while}[n](m) \rrbracket^{\mathbf{MA}}_{\mathsf{Cmd}}(\Sigma', p, \gamma', \sigma') &= \mathsf{iter}(g)(\sigma') \ \mathbf{where} \\
&\quad g : \llbracket \Sigma' \rrbracket^{\mathbf{MA}}_{\mathsf{Sig}} \to \mathsf{L}((1 \times \llbracket \Sigma' \rrbracket^{\mathbf{MA}}_{\mathsf{Sig}}) + \llbracket \Sigma' \rrbracket^{\mathbf{MA}}_{\mathsf{Sig}}) \\
g(\sigma) \ \mathbf{with} \ \sigma[n] & \\
\cdots \mid \mathsf{ff} &= \mathsf{step}^1(\mathsf{ret}_{\mathsf{L}}(\mathsf{inl}(\star, \sigma))) \\
\cdots \mid \mathsf{tt} &= (-, \sigma') \leftarrow_{\mathsf{L}} \llbracket m \rrbracket^{\mathbf{MA}}_{\mathsf{Cmd}}(\Sigma', p, \gamma', \sigma); \mathsf{step}^2(\mathsf{ret}(\mathsf{inr}(\sigma')))
\end{aligned}$$

In the above we have $\Gamma \vdash_{\Sigma} \mathsf{while}[n](m) \div \mathsf{unit}$. The inputs to the interpretation are the future world Σ' (with $p : \Sigma' \geq \Sigma$), the closing substitution for Γ relative to the future world γ', and a semantic store σ'. The interpretation simply iterates the body of the loop while bookkeeping the associated costs.

(3.5.2∗6) *Cost-aware computational adequacy for* ***MA***. The proof for cost-aware adequacy is very similar to that of **STLC**. A particularly important distinction is

[3] If the signature is allowed to range over arbitrary types, then the interpretation of signatures and types become circurlar; unraveling this mutual dependence would require much more complex techniques. Because **MA** is restricted to first-order store we can break the circularity by defining the interpretation of signatures first since they only contain base types.

in the treatment of the lift of the logical relation to commands:

$$\begin{aligned}\mathsf{m}\ \mathsf{cmd}_{\Sigma,A}(R)\ m = \forall \sigma, \sigma' : [\![\Sigma]\!]^{\mathbf{MA}}_{\mathsf{Sig}}, c : \mathbb{N}, a : [\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma).\\ m(\sigma) = \mathsf{step}^c(\mathsf{ret}_{\mathsf{L}}(a, \sigma')) \to\\ \Pi\mu : \mathsf{Store}(\Sigma) \to \mu \sim_\Sigma \sigma \to\\ \Sigma\mathsf{a} : \mathsf{Pg}(\Sigma, A), \mu' : \mathsf{Store}(\Sigma).\\ (\mu, \mathsf{m}) \Downarrow^{\eta\bullet c}_{\P_{\mathsf{E}}/\mathsf{cmd}} (\mu', \mathsf{ret}(\mathsf{a})) \times \mathsf{a}\ R\ a \times \mu' \sim_\Sigma \sigma'\end{aligned}$$

In the above $\mathsf{Store}(\Sigma)$ is the type of stores relative to a signature Σ, R is a relation between closed values of type A relative to Σ (written $\mathsf{Pg}(\Sigma, A)$) and $[\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma)$, and $\mathsf{cmd}_{\Sigma,A}$ lifts R to a relation between closed commands of type A relative to Σ and functions of type $[\![\Sigma]\!]^{\mathbf{MA}}_{\mathsf{Sig}} \to \mathsf{F}([\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma) \times [\![\Sigma]\!]^{\mathbf{MA}}_{\mathsf{Sig}})$ (*i.e.* a semantic command). The relation $\Downarrow^{\eta\bullet c}_{\P_{\mathsf{E}}/\mathsf{cmd}}$ is the phase-separated version of the evaluation relation for commands obtained in a similar fashion to **(3.2.3∗2)**.

Note that the lifted relation universally quantifies over the possible resulting values of the semantic command m, whereas the lifting relation in **(3.4.2∗3)** uses an existential quantification since such a value will always exist. This change is propagated throughout the proof of the fundamental theorem: as mentioned in **(3.5.1∗4)**, while uniqueness of cost bounds is sufficient in the case of the **STLC**, the universal quantification used in the lifting relation for **MA** commands makes it so that we also need to be able to decompose cost bounds.

(3.5.2∗7) Following the recipe for the **STLC**, we obtain the following adequacy theorems for **MA**:

Theorem 3.5.2∗7.1 (Cost-aware adequacy for **MA**)**.** *Let* $\cdot \vdash_\cdot e : \mathsf{bool}$ *be a closed boolean with no free assignables. If* $[\![e]\!]^{\boldsymbol{MA}}_{\mathsf{Exp}} = \mathsf{step}^c(\mathsf{ret}(b))$*, then we have* $e \Downarrow^{\eta\bullet c}_{\P_{\mathsf{E}}} \bar{b}$*. Moreover, let* $\cdot \vdash_\cdot m \div \mathsf{bool}$ *be a closed boolean command with no free assignables. If* $[\![m]\!]^{\boldsymbol{MA}}_{\mathsf{Cmd}} = \mathsf{step}^c(\mathsf{ret}((b, \star)))$*, then we have* $(\cdot, m) \Downarrow^{\eta\bullet c}_{\P_{\mathsf{E}}/\mathsf{cmd}} (\cdot, \bar{b})$*.*

Theorem 3.5.2∗7.2 (Extensional adequacy for **MA**)**.** *Suppose* $u : \P_{\mathsf{E}}$*. Let* $\cdot \vdash_\cdot e : \mathsf{bool}$ *be a closed boolean with no free assignables. If* $[\![e]\!]^{\boldsymbol{MA}}_{\mathsf{Exp}} = \mathsf{ret}(b)$*, then we have* $e \Downarrow \bar{b}$*. Moreover, let* $\cdot \vdash_\cdot m \div \mathsf{bool}$ *be a closed boolean command with no free assignables. If* $[\![m]\!]^{\boldsymbol{MA}}_{\mathsf{Cmd}} = \mathsf{ret}((b, \star))$*, then we have* $(\cdot, m) \Downarrow_{\mathsf{cmd}} (\cdot, \bar{b})$*.*

3.5.3. Models of calf$^\omega$.

(3.5.3∗1) Models of **calf**$^\omega$ may be obtained by extending the models of **calf***. The lift operation is interpreted using a monad that models partiality; for concreteness and convenience we choose the quotient inductive-inductive partiality monad of Altenkirch, Danielsson, and Kraus [ADK17]. Because the iteration operation in **MA** may be seen as a continuous functional we may implement it as a fixed-point.

3.6. CASE STUDY: PCF

(3.6∗1) One of the primary features that distinguishes **MA** presented in Section 3.5 and **PCF** is general fixed-points. While the iteration mechanism of **MA** can be defined as a fixed-point (and we define it this way in the model of $\mathbf{calf}^{\omega}$), the converse does not hold. Therefore, to define a denotational semantics for **PCF** in the same synthetic fashion as **STLC** and **MA**, I propose to extend $\mathbf{calf}^{\star}$ by axiomatizing the existence of fixed-points for endomaps between (the lifting of) types of a universe of *predomains*. The resulting theory, dubbed $\mathbf{calf}^{\infty}$, can then be seen as an extension of $\mathbf{calf}^{\omega}$.

(3.6∗2) Before proposing the theory $\mathbf{calf}^{\infty}$, I briefly describe the kind of models that I have in mind. A natural option is to interpret $\mathbf{calf}^{\infty}$ in a model of *synthetic domain theory* (SDT). As the name suggests, SDT diverges from classic domain theory in that one may construct maps of domains using ordinary (intuitionistic) set-theoretic language without the burden of checking continuity conditions. More details about SDT may be found in the references [FR97; Hyl91; Reu95; RS99].

(3.6∗3) A topos that models SDT is equipped with a class of "computational" truth values Σ called a *dominance* that determines the lifting monad L that is used to classify partial computations. The initial algebra ω and final coalgebra $\overline{\omega}$ of L then represent synthetic versions of the *generic* $\mathbb{N}$-chain and $\mathbb{N}$-chain equipped with a point at infinity, respectively. One way to obtain a nice class of predomains is via *well complete* types. We say a type A is *complete* when any ω-chain in A is uniquely extended to an $\overline{\omega}$-chain. The class of predomains then consists of the well complete types, which is any type A such that $\mathsf{L}(A)$ is complete. Classically, a *domain* is a predomain with a least element; in the synthetic setting this corresponds to equipping predomains with an L-algebra structure. Importantly, *any* endomap of domains has a fixed-point.

(3.6∗4) Predomains enjoy a number of closure conditions, and one may choose predomains so that they are closed under the type structures of **PCF**. Given the considerations in **(3.6∗3)**, we propose to axiomatize a universe of predomains in $\mathbf{calf}^{\infty}$ whose types support general recursion/partiality. Because we need to account for the cost of computations in predomains, we introduce a pair of universes:

$$\begin{aligned}\mathsf{PreDom}^{+} &: \mathsf{tp}^{+}\\ \mathsf{ElDom}^{+} &: \mathsf{tm}^{+}(\mathsf{PreDom}^{+}) \to \mathsf{tp}^{+}\\ \mathsf{PreDom}^{\ominus} &: \mathsf{tp}^{\ominus}\\ \mathsf{ElDom}^{\ominus} &: \mathsf{tm}^{\ominus}(\mathsf{PreDom}^{\ominus}) \to \mathsf{tp}^{\ominus}\end{aligned}$$

At minimum we need to close predomains under natural numbers, lifting, and functions. As an example, closure under lifting may be specified as follows:

$$\begin{aligned}&\mathsf{L} : \mathsf{tp}^{+} \to \mathsf{tp}^{\ominus}\\ &\widehat{\mathsf{L}} : \mathsf{tm}^{+}(\mathsf{PreDom}^{+}) \to \mathsf{tm}^{\ominus}(\mathsf{PreDom}^{\ominus})\\ &- : \{\widehat{A}\}\ \mathsf{ElDom}^{\ominus}(\widehat{\mathsf{L}}(\widehat{A})) = \mathsf{L}(\mathsf{ElDom}^{+}(\widehat{A}))\end{aligned}$$

General recursion is available for lifted predomains:

$$\mathsf{fix} : \{\widehat{A}\}\ \mathsf{U}(\mathsf{U}(\mathsf{L}(\mathsf{ElDom}^{+}(\widehat{A}))) \to \mathsf{L}(\mathsf{ElDom}^{+}(\widehat{A}))) \to \mathsf{L}(\mathsf{ElDom}^{+}(\widehat{A}))$$

3.6.1. Cost-aware denotational semantics of PCF.

(3.6.1∗1) Following the case studies in Sections 3.4 and 3.5, I propose to define a denotational semantics for **PCF** in **calf**$^{\infty}$. As for the **STLC** and **MA**, I will work with a call-by-value operational semantics (see Fig. 3.3). Write $\mathsf{Ty}^{\mathbf{PCF}}$ and $\mathsf{Tm}^{\mathbf{PCF}}(\Gamma, A)$ for the type of **PCF** types and terms, respectively. Following the standard call-by-push-value decomposition in predomains, types of **PCF** are interpreted as value predomains PreDom^{+} and terms are interpreted as elements of lifted predomains:

$$\begin{aligned}[\![-]\!]_{\mathsf{Ty}} &: \mathsf{Ty}^{\mathbf{PCF}} \to \mathsf{ElDom}^{+}(\mathsf{PreDom}^{+})\\ [\![-]\!] &: \{\Gamma, A\}\ \mathsf{Tm}^{\mathbf{PCF}}(\Gamma, A) \to [\![\Gamma]\!]_{\mathsf{Ty}} \to \mathsf{L}([\![A]\!]_{\mathsf{Ty}})\end{aligned}$$

(3.6.1∗2) I believe that cost-aware computational adequacy for **PCF** may be proved using a logical relations proof as in the case for **STLC** and **MA**. Similar to the truncation axiom for iteration **(3.5.1∗3)** for **MA**, it is expected that one needs to axiomatize in **calf**$^{\infty}$ similar properties that hold of either classical or synthetic domains to carry out the adequacy theorem.

$$\frac{\Gamma, x : A_1 \to A_2, y : A_1 \vdash e : A_2}{\Gamma \vdash \mathsf{fun}(x.y.\ e) : A_1 \to A_2}$$

Figure 3.3: Call-by-value **PCF**, selected typing rules.

3.6.2. Models of calf$^\infty$.

(3.6.2∗1) For the models of **calf**$^\infty$, the plan is to construct a version of the counting model (see Section 2.2.10) in an SDT topos $\mathbf{X}$. The interpretation of predomains and lifting should resemble SDT-topos models of programming languages with recursion [Sim99; Sim04] with modifications to account for the interaction of cost structure and partiality. As for **calf**$^\omega$, a good starting point for the interpretation of the cost-aware lift monad is just the composition of the ordinary lifting monad with the cost monad: $\mathsf{L}(A) \triangleq \Sigma(p : \Sigma).\ p \to (\bullet\mathbb{C} \times A)$. Naturally PreDom^+ is interpreted as predomains of $\mathbf{X}$, and $\mathsf{PreDom}^\ominus$ is interpreted as the type of algebras for the cost monad valued in predomains. The required closure conditions on both should follow from the properties of predomains.

(3.6.2∗2) An anticipated problem is the relationship between free computations and computations in a predomain. In contrast to **calf**$^\omega$, not all free computations may be considered the computations of a predomain since predomains are not closed under arbitrary types of **calf**. Although this should not pose a problem for studying **PCF**, it would be good to understand this limitation in the context of **calf**$^\infty$ as a general purpose programming language. For instance, how should one extend the theory of cost bounds (see Section 2.2.6) to partial computations?

(3.6.2∗3) *Instantiating the model construction.* One way to obtain an SDT topos is given in Sterling and Harper [SH22]. Here *op. cit.* first define a concrete category $\mathscr{C}$ complying with *axiomatic domain theory* [Fio94] internal to a presheaf topos for *information flow*, and then extend $\mathscr{C}$ to a model of SDT using an adaption of the result of Fiore and Plotkin [FP96]. Because the counting model of **calf** is a simpler version of the information flow topos, I expect that it would be possible to reuse these results in the context of interpreting **calf**$^\infty$.

3.7. TIMELINE

(3.7∗1) I expect that the proposed work can be completed within one year. The metatheory of **calf** and cost-aware denotational semantics of **PCF** may be broken down into the following tasks, both in order of perceived difficulty:

1. Axiomatize STC for **calf** and construct the computability algebra.
2. Construct an STC topos for canonicity and extract the computability result.
3. Give a complete definition of $\mathbf{calf}^{\infty}$.
4. Construct an adequate denotational model of **PCF** in $\mathbf{calf}^{\infty}$.
5. Interpret $\mathbf{calf}^{\infty}$ using SDT.
6. Instantiate the model at a concrete SDT topos.

BIBLIOGRAPHY

[1] S. F. Allen et al. “Innovations in computational type theory using Nuprl”. In: *Journal of Applied Logic* 4.4 (2006). Towards Computer Aided Mathematics, pp. 428–469. ISSN: 1570-8683.

[2] Stuart Frazier Allen. “A non-type-theoretic semantics for type-theoretic language”. PhD thesis. Ithaca, NY, USA: Cornell University, 1987.

[3] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. “Partiality, Revisited: The Partiality Monad as a Quotient Inductive-Inductive Type”. In: *Foundations of Software Science and Computation Structures*. Ed. by Javier Esparza and Andrzej S. Murawski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 534–549. ISBN: 978-3-662-54458-7. DOI: 10.1007/978-3-662-54458-7_31. arXiv: 1610.09254 [cs.LO].

[4] Thorsten Altenkirch and Ambrus Kaposi. “Normalisation by Evaluation for Dependent Types”. In: *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Ed. by Delia Kesner and Brigitte Pientka. Vol. 52. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 6:1–6:16. ISBN: 978-3-95977-010-1. DOI: 10.4230/LIPIcs.FSCD.2016.6. URL: http://drops.dagstuhl.de/opus/volltexte/2016/5972.

[5] Carlo Angiuli. “Computational Semantics of Cartesian Cubical Type Theory”. PhD thesis. Carnegie Mellon University, 2019.

[6] Guy Blelloch and John Greiner. “Parallelism in Sequential Functional Languages”. In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 226–237. ISBN: 0-89791-719-7. DOI: 10.1145/224164.224210.

[7] Ana Bove and Venanzio Capretta. “Modelling general recursion in type theory”. In: *Mathematical Structures in Computer Science* 15.4 (2005), pp. 671–708. DOI: 10.1017/S0960129505004822.

[8] Ezgi Çiçek et al. “Relational Cost Analysis”. In: *POPL 2017: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Paris, France: Association for Computing Machinery, 2017, pp. 316–329. ISBN: 978-1-4503-4660-3. DOI: `10.1145/3009837.3009858`.

[9] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. “The Taming of the Rew: A Type Theory with Computational Assumptions”. In: *Proceedings of the ACM on Programming Languages*. POPL 2021 (2021). URL: `https://hal.archives-ouvertes.fr/hal-02901011`.

[10] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986. ISBN: 0-13-451832-2.

[11] Thierry Coquand. “Canonicity and normalization for dependent type theory”. In: *Theoretical Computer Science* 777 (2019). In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I, pp. 184–191. ISSN: 0304-3975. DOI: `10.1016/j.tcs.2019.01.015`. arXiv: `1810.09367 [cs.PL]`.

[12] Nils Anders Danielsson. “Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 133–144. ISBN: 9781595936899. DOI: `10.1145/1328438.1328457`. URL: `https://doi.org/10.1145/1328438.1328457`.

[13] Rowan Davies and Frank Pfenning. “A Modal Analysis of Staged Computation”. In: *Journal of the ACM* 48 (Sept. 1999). DOI: `10.1145/382780.382785`.

[14] Peter Dybjer and Anton Setzer. “Indexed Induction-Recursion”. In: *The Journal of Logic and Algebraic Programming* 66 (June 2004). DOI: `10.1016/j.jlap.2005.07.001`.

[15] Marcelo Fiore. “Axiomatic Domain Theory in Categories of Partial Maps”. PhD thesis. University of Edinburgh, Nov. 1994. URL: `https://era.ed.ac.uk/handle/1842/406`.

[16] Marcelo P. Fiore and Gordon D. Plotkin. “An Extension of Models of Axiomatic Domain Theory to Models of Synthetic Domain Theory”. In: *Computer Science Logic, 10th International Workshop, CSL ’96, Annual Conference of the EACSL, Utrecht, The Netherlands, September 21-27, 1996, Selected Papers*. Ed. by Dirk van Dalen and Marc Bezem. Vol. 1258. Lecture Notes in Computer Science. Springer, 1996, pp. 129–149. DOI: `10.1007/3-540-63172-0\_36`.

[17] Marcelo P. Fiore and Giuseppe Rosolini. “Two models of synthetic domain theory”. In: *Journal of Pure and Applied Algebra* 116.1 (1997), pp. 151–162. ISSN: 0022-4049. DOI: `10.1016/S0022-4049(96)00164-8`.

[18] Gottlob Frege. “Sense and Reference”. In: *The Philosophical Review* 57.3 (1948), pp. 209–230. ISSN: 00318108, 15581470. URL: `http://www.jstor.org/stable/2181485`.

[19] Daniel Gratzer. *Normalization for multimodal type theory.* 2021. arXiv: `2106.01414 [cs.LO]`.

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

[21] Martin A. T. Handley, Niki Vazou, and Graham Hutton. “Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019). DOI: `10.1145/3371092`.

[22] Robert Harper. “Constructing Type Systems over an Operational Semantics”. In: *Journal of Symbolic Computation* 14.1 (July 1992), pp. 71–84. ISSN: 0747-7171.

[23] Robert Harper. ***PFPL*** *Supplement: PCF By Value.* May 11, 2020. URL: `http://www.cs.cmu.edu/~rwh/pfpl/supplements/pcfv.pdf`.

[24] Robert Harper. ***PFPL*** *Supplement: Types and Parallelism.* 2018. URL: `https://www.cs.cmu.edu/~rwh/pfpl/supplements/par.pdf`.

[25] Robert Harper. *Practical Foundations for Programming Languages.* First. New York, NY, USA: Cambridge University Press, 2012.

[26] Robert Harper. *Practical Foundations for Programming Languages.* Second. New York, NY, USA: Cambridge University Press, 2016.

[27] Robert Harper, John C. Mitchell, and Eugenio Moggi. “Higher-Order Modules and the Phase Distinction”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* San Francisco, California, USA: Association for Computing Machinery, 1990, pp. 341–354. ISBN: 0-89791-343-4. DOI: `10.1145/96709.96744`.

[28] Jan Hoffmann. *Cost semantics.* Unpublished. 2019.

[29] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. “Resource Aware ML”. In: *Computer Aided Verification.* Ed. by P. Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 781–786. ISBN: 978-3-642-31424-7.

[30] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. “Towards Automatic Resource Bound Analysis for OCaml”. In: *POPL 2017: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* Paris, France: Association for Computing Machinery, 2017, pp. 359–373. ISBN: 978-1-4503-4660-3. URL: `10.1145/3009837.3009842`.

[31] Douglas J. Howe. “Equality in Lazy Computation Systems”. In: *Proceedings of Fourth IEEE Symposium on Logic in Computer Science.* New York: IEEE Computer Society, 1989, pp. 198–203.

[32] J. M. E. Hyland. “First steps in synthetic domain theory”. In: *Category Theory.* Ed. by Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 131–156. ISBN: 978-3-540-46435-8.

[33] G. Kahn. “Natural semantics”. In: *STACS 87.* Ed. by Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 22–39. ISBN: 978-3-540-47419-7.

[34] G. A. Kavvos. “On the Semantics of Intensionality and Intensional Recursion”. PhD thesis. 2017. arXiv: `1712.09302`.

[35] G. A. Kavvos et al. “Recurrence Extraction for Functional Programs through Call-by-Push-Value”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019). DOI: `10.1145/3371083`.

[36] S. C. Kleene. “Recursive predicates and quantifiers”. In: *Trans. Amer. Math. Soc.* 53 (1943), pp. 41–73. ISSN: 0002-9947. DOI: `10.2307/1990131`. URL: `https://doi.org/10.2307/1990131`.

[37] F. William Lawvere. “Functorial Semantics of Algebraic Theories”. PhD thesis. Columbia University, 1963.

[38] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2).* Norwell, MA, USA: Kluwer Academic Publishers, 2004. ISBN: 1-4020-1730-8.

[39] Per Martin-Löf. “Constructive Mathematics and Computer Programming”. In: *6th International Congress for Logic, Methodology and Philosophy of Science.* Published by North Holland, Amsterdam. 1982. Hanover, Aug. 1979, pp. 153–175.

[40] Rasmus Ejlers Møgelberg and Marco Paviotti. “Denotational Semantics of Recursive Types in Synthetic Guarded Domain Theory”. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science.* New York, NY, USA: Association for Computing Machinery, 2016, pp. 317–326. ISBN: 978-1-4503-4391-6. DOI: `10.1145/2933575.2934516`.

[41] Eugenio Moggi. “Notions of computation and monads”. In: *Information and Computation* 93.1 (1991). Selections from 1989 IEEE Symposium on Logic in Computer Science, pp. 55–92. ISSN: 0890-5401. DOI: `10.1016/0890-5401(91)90052-4`.

[42] Yue Niu and Robert Harper. *Cost-Aware Type Theory.* 2020. arXiv: `2011.03660 [cs.PL]`.

[43] Yue Niu et al. “A Cost-Aware Logical Framework”. In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022). DOI: `10.1145/3498670`. arXiv: `2107.04663 [cs.PL]`.

[44] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. “A Model of PCF in Guarded Type Theory”. In: *Electronic Notes in Theoretical Computer Science* 319.Supplement C (2015). The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI), pp. 333–349. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2015.12.020`.

[45] Pierre-Marie Pédrot and Nicolas Tabareau. “The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019). DOI: `10.1145/3371126`.

[46] G.D. Plotkin. “LCF considered as a programming language”. In: *Theoretical Computer Science* 5.3 (1977), pp. 223–255. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(77)90044-5`. URL: `https://www.sciencedirect.com/science/article/pii/0304397577900445`.

[47] Gordon D Plotkin. *A structural approach to operational semantics.* Aarhus university, 1981.

[48] *POPL 2017: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* Paris, France: Association for Computing Machinery, 2017. ISBN: 978-1-4503-4660-3.

[49] Vineet Rajani et al. "A Unifying Type-Theory for Higher-Order (Amortized) Cost Analysis". In: *Proceedings of the ACM on Programming Languages* 5.POPL (Jan. 2021). DOI: `10.1145/3434308`.

[50] Bernhard Reus. "Program Verification in Synthetic Domain Theory". PhD thesis. München: Ludwig-Maximilians-Universität München, Nov. 1995.

[51] Bernhard Reus and Thomas Streicher. "General synthetic domain theory — a logical approach". In: *Mathematical Structures in Computer Science* 9.2 (1999), pp. 177–223. DOI: `10.1017/S096012959900273X`.

[52] Alex Simpson. "Computational adequacy for recursive types in models of intuitionistic set theory". In: *Annals of Pure and Applied Logic* 130.1 (2004). Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS), pp. 207–275. ISSN: 0168-0072. DOI: `https://doi.org/10.1016/j.apal.2003.12.005`. URL: `https://www.sciencedirect.com/science/article/pii/S0168007204000624`.

[53] Alex K. Simpson. "Computational Adequacy in an Elementary Topos". In: *Computer Science Logic*. Ed. by Georg Gottlob, Etienne Grandjean, and Katrin Seyr. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 323–342. ISBN: 978-3-540-48855-2.

[54] Alex K. Simpson and Giuseppe Rosolini. "Using Synthetic Domain Theory to Prove Operational Properties of a Polymorphic Programming Language Based on Strictness". In: 2004.

[55] Jonathan Sterling. 2020.

[56] Jonathan Sterling. "First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory". Version 1.1, revised May 2022. PhD thesis. Carnegie Mellon University, 2021. DOI: `10.5281/zenodo.6990769`.

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

[58] Jonathan Sterling and Robert Harper. "Guarded Computational Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. Oxford, United Kingdom: Association for Computing Machinery, 2018. ISBN: 978-1-4503-5583-4. arXiv: `1804.09098 [cs.LO]`.

[59] Jonathan Sterling and Robert Harper. “Logical Relations as Types: Proof-Relevant Parametricity for Program Modules”. In: *Journal of the ACM* 68.6 (Oct. 2021). ISSN: 0004-5411. DOI: `10.1145/3474834`. arXiv: `2010.08599 [cs.PL]`.

[60] Jonathan Sterling and Robert Harper. “Sheaf semantics of termination-insensitive noninterference”. In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Ed. by Amy P. Felty. Vol. 228. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Aug. 2022, 5:1–5:19. ISBN: 978-3-95977-233-4. DOI: `10.4230/LIPIcs.FSCD.2022.5`. arXiv: `2204.09421 [cs.PL]`.

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

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

[63] Peng Wang, Di Wang, and Adam Chlipala. “TiML: A Functional Language for Practical Complexity Analysis with Invariants”. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017). DOI: `10.1145/3133903`.

.1. COMPLETE DEFINITION OF calf

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

Figure .4: Core ∂**cbpv** calculus.

$$
\begin{aligned}
\mathsf{bind/ret} &: \{A, X\}\ (a : \mathsf{tm}^{+}(A)) \to (f : \mathsf{tm}^{+}(A) \to \mathsf{tm}^{\ominus}(X)) \to \mathsf{bind}(\mathsf{ret}(a); f) = f(a) \\
\mathsf{tbind/ret} &: \{A\}\ (a : \mathsf{tm}^{+}(A)) \to (f : \mathsf{tm}^{+}(A) \to \mathsf{tp}^{\ominus}) \to \mathsf{tbind}(\mathsf{ret}(a); f) = f(a) \\
\mathsf{dbind/ret} &: \{A, X\}\ (a : \mathsf{tm}^{+}(A)) \to (f : a : \mathsf{tm}^{+}(A) \to \mathsf{tm}^{\ominus}(X(a))) \to \mathsf{dbind}(\mathsf{ret}(a); f) = f(a) \\
\mathsf{bind/assoc} &: \{A, B, X\}\ (e : \mathsf{tm}^{\ominus}(\mathsf{F}(A))) \to \\
&\quad (f : \mathsf{tm}^{+}(A) \to \mathsf{tm}^{\ominus}(\mathsf{F}(B))) \to (g : \mathsf{tm}^{+}(B) \to \mathsf{tm}^{\ominus}(C)) \to \\
&\quad \mathsf{bind}(\mathsf{bind}(e; f); g) = \mathsf{bind}(e; \lambda a.\, \mathsf{bind}(f(a); g)) \\
\mathsf{tbind/assoc} &: \{A, B, X\}\ (e : \mathsf{tm}^{\ominus}(\mathsf{F}(A))) \to (f : \mathsf{tm}^{+}(A) \to \mathsf{tm}^{\ominus}(\mathsf{F}(B))) \to \\
&\quad \mathsf{tbind}(\mathsf{bind}(e; f); X) = \mathsf{tbind}(e; \lambda a.\, \mathsf{tbind}(f(a); X))
\end{aligned}
$$

Figure .5: Computation and associativity laws for sequencing.

$$\begin{aligned}
\mathbb{C} &: \mathbf{Jdg} \\
0 &: \mathbb{C} \\
+ &: \mathbb{C} \to \mathbb{C} \to \mathbb{C} \\
\leq &: \mathbb{C} \to \mathbb{C} \to \mathbf{Jdg} \\
\mathsf{costMon} &: \mathsf{isCostMonoid}(\mathbb{C}, 0, +, \leq)
\end{aligned}$$

$$\begin{aligned}
\widehat{\mathbb{C}} &: \mathsf{tp}^{\ominus} \\
(\mathsf{out}_{\mathbb{C}}, \mathsf{in}_{\mathbb{C}}) &: \mathsf{tm}^{\ominus}(\widehat{\mathbb{C}}) \cong \mathbb{C} \\
\widehat{\leq} &: \widehat{\mathbb{C}} \to \widehat{\mathbb{C}} \to \mathsf{tp}^{\ominus} \\
(\mathsf{out}_{\leq}, \mathsf{in}_{\leq}) &: \{c, c'\}\ \mathsf{tm}^{\ominus}(c \mathrel{\widehat{\leq}} c') \cong (\mathsf{out}_{\mathbb{C}}(c) \leq \mathsf{out}_{\mathbb{C}}(c'))
\end{aligned}$$

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

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

Figure .6: Cost structure and cost effect.

$$
\begin{aligned}
\P_{\mathsf{E}} &: \mathbf{Jdg}\\
\P_{\mathsf{E}}/\mathsf{uni} &: \{u, v : \P_{\mathsf{E}}\}\ u = v\\
\mathsf{ext}A &:= \P_{\mathsf{E}} \to A\\
\mathsf{step}/\P_{\mathsf{E}} &: \{X, e, c\}\ \mathsf{extstep}^c(e) = e\\
\bigcirc^+ &: \mathsf{tp}^+ \to \mathsf{tp}^+\\
\_ &: \{A\}\ \mathsf{tm}^+(\bigcirc^+ A) \cong \bigcirc(\mathsf{tm}^+(A))\\
\\
\bullet &: \mathsf{tp}^+ \to \mathsf{tp}^+\\
\eta_\bullet &: \mathsf{tm}^+(A) \to \mathsf{tm}^+(\bullet A)\\
* &: \P_{\mathsf{E}} \to \mathsf{tm}^+(\bullet A)\\
\_ &: \Pi a : \mathsf{tm}^+(A).\, \Pi u : \P_{\mathsf{E}}.\, \eta_\bullet(a) = *(u)\\
\mathsf{ind}_\bullet &: \{A\}\ (a : \mathsf{tm}^+(\bullet A)) \to (X : \mathsf{tm}^+(\bullet A) \to \mathsf{tp}^\ominus) \to\\
&\quad (x_0 : (a : \mathsf{tm}^+(A)) \to \mathsf{tm}^\ominus(X(\eta_\bullet(a)))) \to\\
&\quad (x_1 : (u : \P_{\mathsf{E}}) \to \mathsf{tm}^\ominus(X(*(u)))) \to\\
&\quad ((a : \mathsf{tm}^+(A)) \to (u : \P_{\mathsf{E}}) \to x_0(a) = x_1(u)) \to\\
&\quad \mathsf{tm}^\ominus(X(a))\\
\mathsf{ind}_\bullet/\eta &: \{A\}\ (a : \mathsf{tm}^+(A)) \to (X : \mathsf{tm}^+(\bullet A) \to \mathsf{tp}^\ominus) \to\\
&\quad (x_0 : (a : \mathsf{tm}^+(A)) \to \mathsf{tm}^\ominus(X(\eta_\bullet(a)))) \to\\
&\quad (x_1 : (u : \P_{\mathsf{E}}) \to \mathsf{tm}^\ominus(X(*(u)))) \to\\
&\quad (h : (a : \mathsf{tm}^+(A)) \to (u : \P_{\mathsf{E}}) \to x_0(a) = x_1(u)) \to\\
&\quad \mathsf{ind}_\bullet(\eta_\bullet(a), X, x_0, x_1, h) = x_0(a)\\
\mathsf{ind}_\bullet/* &: \{A\}\ (u : \P_{\mathsf{E}}) \to (X : \mathsf{tm}^+(\bullet A) \to \mathsf{tp}^\ominus) \to\\
&\quad (x_0 : (a : \mathsf{tm}^+(A)) \to \mathsf{tm}^\ominus(X(\eta_\bullet(a)))) \to\\
&\quad (x_1 : (u : \P_{\mathsf{E}}) \to \mathsf{tm}^\ominus(X(*(u)))) \to\\
&\quad (h : (a : \mathsf{tm}^+(A)) \to (u : \P_{\mathsf{E}}) \to x_0(a) = x_1(u)) \to\\
&\quad \mathsf{ind}_\bullet(*(u), X, x_0, x_1, h) = x_1(u)
\end{aligned}
$$

Figure .7: Modal account of the phase distinction.

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

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

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

$$\begin{aligned}
1 &: \mathsf{tp}^{+} \\
\star &: \mathsf{tm}^{+}(1) \\
\eta_1 &: \{u, v\}\ u =_{\mathsf{tm}^{+}(1)} v
\end{aligned}$$

$$\begin{aligned}
+ &: \mathsf{tp}^{+} \to \mathsf{tp}^{+} \to \mathsf{tp}^{+} \\
\mathsf{inl} &: \{A, B\}\ \mathsf{tm}^{+}(A) \to \mathsf{tm}^{+}(A + B) \\
\mathsf{inr} &: \{A, B\}\ \mathsf{tm}^{+}(B) \to \mathsf{tm}^{+}(A + B) \\
\mathsf{case} &: \{A, B\}\ (s : \mathsf{tm}^{+}(A + B)) \to (\mathsf{tm}^{+}(A + B) \to \mathsf{tp}^{\ominus}) \to \\
&\quad ((a : \mathsf{tm}^{+}(A)) \to \mathsf{tm}^{\ominus}(X(\mathsf{inl}(a)))) \to \\
&\quad ((b : \mathsf{tm}^{+}(B)) \to \mathsf{tm}^{\ominus}(X(\mathsf{inl}(b)))) \to \mathsf{tm}^{\ominus}(X(s)) \\
\mathsf{case}_{\mathsf{inl}} &: \{A, B, X, e_0, e_1\}\ (a : \mathsf{tm}^{+}(A)) \to \mathsf{case}(\mathsf{inl}(a); X; e_0; e_1) = e_0(a) \\
\mathsf{case}_{\mathsf{inr}} &: \{A, B, X, e_0, e_1\}\ (b : \mathsf{tm}^{+}(B)) \to \mathsf{case}(\mathsf{inr}(b); X; e_0; e_1) = e_1(b)
\end{aligned}$$

$$\begin{aligned}
\mathsf{nat} &: \mathsf{tp}^{+} \\
\mathsf{zero} &: \mathsf{tm}^{+}(\mathsf{nat}) \\
\mathsf{suc} &: \mathsf{tm}^{+}(\mathsf{nat}) \to \mathsf{tm}^{+}(\mathsf{nat}) \\
\mathsf{rec} &: (n : \mathsf{tm}^{+}(\mathsf{nat})) \to (X : \mathsf{tm}^{+}(\mathsf{nat}) \to \mathsf{tp}^{\ominus}) \to \mathsf{tm}^{\ominus}(X(\mathsf{zero})) \to \\
&\quad ((n : \mathsf{tm}^{+}(\mathsf{nat})) \to \mathsf{tm}^{\ominus}(X(n)) \to \mathsf{tm}^{\ominus}(X(\mathsf{suc}(n)))) \to \mathsf{tm}^{\ominus}(X(n)) \\
\mathsf{rec/zero} &: \{X, e_0, e_1\}\ \mathsf{rec}(\mathsf{zero}; X; e_0; e_1) = e_0 \\
\mathsf{rec/suc} &: \{n, X, e_0, e_1\}\ \mathsf{rec}(\mathsf{suc}(n); X; e_0; e_1) = e_1(n)(\mathsf{rec}(n; X; e_0; e_1))
\end{aligned}$$

Figure .8: Types

$$\begin{aligned}
\mathsf{L} &: \mathbb{C} \to \mathsf{tp}^+ \to \mathsf{tp}^+ \\
\mathsf{nil} &: \{c, A\}\ \mathsf{L}^c(A) \\
\mathsf{cons} &: \{c, A\}\ \mathsf{tm}^+(A) \to \mathsf{tm}^+(\mathsf{L}^c(A)) \to \mathsf{tm}^+(\mathsf{L}^c(A)) \\
\mathsf{rec} &: \{c, A\}\ (l : \mathsf{tm}^+(\mathsf{L}^c(A))) \to (X : \mathsf{tm}^+(\mathsf{L}^c(A)) \to \mathsf{tp}^\ominus) \to \\
&\quad (\mathsf{tm}^\ominus(X(\mathsf{nil}))) \to \\
&\quad ((a : \mathsf{tm}^+(A)) \to (l : \mathsf{tm}^+(\mathsf{L}^c(A))) \to \mathsf{tm}^\ominus(X(l)) \to \mathsf{tm}^\ominus(X(\mathsf{cons}(a; l)))) \to \\
&\quad \mathsf{tm}^\ominus(X(l)) \\
\mathsf{rec/nil} &: \{c, A, X, e_0, e_1\}\ \mathsf{rec}_{\mathsf{L}}(\mathsf{nil}; X; e_0; e_1) = e_0 \\
\mathsf{rec/cons} &: \{c, A, a, X, e_0, e_1\}\ (l : \mathsf{tm}^+(\mathsf{L}^c(A))) \to \\
&\quad \mathsf{rec}_{\mathsf{L}}(\mathsf{cons}(a; l); X; e_0; e_1) = \mathsf{step}^c(e_1(a)(l)(\mathsf{rec}_{\mathsf{L}}(l; X; e_0; e_1)))
\end{aligned}$$

Figure .9: Types, continued

$$\begin{aligned}
\mathsf{lam}_{\mathsf{step}} &: \{A, X, c\}\ (f : (a : \mathsf{tm}^+(A)) \to \mathsf{tm}^\ominus(X(a))) \to \mathsf{step}^c(\mathsf{lam}(f)) = \mathsf{lam}(\mathsf{step}^c(f)) \\
\mathsf{pair}^{+-}_{\mathsf{step}} &: \{A, X, c\}\ (e : \Sigma(\mathsf{tm}^+(A))(\lambda a.\, \mathsf{tm}^\ominus(X(a)))) \to \mathsf{step}^c(e) = (e \cdot 1, \mathsf{step}^c(e \cdot 2)) \\
\mathsf{case}_{\mathsf{step}} &: \{A, B, X, e_0, e_1, c\}\ (s : \mathsf{tm}^+(A + B)) \to \\
&\quad \mathsf{step}^c(\mathsf{case}(s; X; e_0; e_1)) = \mathsf{case}(s; X; \lambda a.\, \mathsf{step}^c(e_0(a)); \lambda b.\, \mathsf{step}^c(e_1(b)))
\end{aligned}$$

Figure .10: Interaction of **step** with type structure.

$$\begin{aligned}
\& &: \{A, B : \mathsf{tp}^+\}\ \mathsf{tm}^\ominus(\mathsf{F}(A)) \to \mathsf{tm}^\ominus(\mathsf{F}(B)) \to \mathsf{tm}^\ominus(\mathsf{F}(A \times B)) \\
\&_{\mathsf{join}} &: \{A, B, c_1, c_2, a, b\}\ (\mathsf{step}^{c_1}(\mathsf{ret}(a)))\ \&\ (\mathsf{step}^{c_2}(\mathsf{ret}(b))) = \mathsf{step}^{c_1 \otimes c_2}(\mathsf{ret}((a, b)))
\end{aligned}$$

Figure .11: Parallelism.