A metalanguage for cost-aware denotational semantics

Yue Niu
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
yuen@cs.cmu.edu

Robert Harper
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
rwh@cs.cmu.edu

2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) | 979-8-3503-3587-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/LICS56636.2023.10175777

***Abstract*—We present metalanguages for developing synthetic cost-aware denotational semantics of programming languages. Extending recent advances by Niu et al. in cost and behavioral verification in dependent type theory, we define two successively more expressive metalanguages for studying cost-aware metatheory. We construct synthetic denotational models of the simply-typed lambda calculus and Modernized Algol, a language with first-order store and while loops, and show that they satisfy a cost-aware generalization of the classic Plotkin-type computational adequacy theorem. Moreover, by developing our proofs in a synthetic language of phase-separated constructions of intension and extension, our results easily restrict to the corresponding extensional theorems. Consequently, our work provides a positive answer to the conjecture raised in op. cit. and contributes a framework for cost-aware programming, verification, and metatheory.**

***Index Terms*—types, semantics, cost analysis**

I. Introduction

Denotational semantics is a well-established method for obtaining an *equational theory* for program verification. Whereas the operational semantics of a programming language gives meaning to programs via closed, whole program computation, denotational semantics aims to assign a *compositional* theory to *open* programs amenable to equational/algebraic reasoning. A well-behaved denotational semantics respects the operational meaning of programs in the sense that the denotation of a program is invariant under evaluation. This property is known as *soundness*. Conversely, for a denotational model to be useful, it must be conservative enough as an equational theory so that equations in the denotational model can be reflected in the operational semantics. This is known as *computational adequacy*.[1] Denotational semantics satisfying these properties have been studied for a long time, starting with Plotkin's work on the investigation of LCF as a programming language (PCF) [1].

Although the question of computational adequacy has been traditionally studied in the context of denotational semantics, recent work on program cost analysis in type theory has broached the possibility of viewing adequacy in the more general context of equational theories. In particular, Niu et al. proposed a dependent type theory **calf** (**c**ost-**a**ware **l**ogical **f**ramework) that provides a rich specification language supporting both behavioral and cost verification of functional programs. That work formalizes a myriad of case studies of the cost analysis of algorithms in the framework and proves the consistency of **calf** via a model construction. As a type theory, **calf** can be thought of as the semantic domain of a denotational semantics in the sense that it furnishes an equational theory for program analysis. Moreover, as a cost analysis framework, **calf** does not stipulate a cost semantics for programs; instead, the users of the framework is responsible for specifying the cost model of the algorithms they define. This raises a natural question: how does one know if a cost model is reasonable relative to a given programming language? In the concluding remarks, the authors conjecture that the choice of a cost model with respect to an operational semantics may be justified by an *internal* adequacy theorem in the style of Plotkin.

In this paper, we substantiate this idea and develop extensions of **calf** that promote it to a metalanguage for *synthetic cost-aware denotational semantics*. To illustrate our approach, we first define **calf***, an extension of **calf** with universes and inductive types, which we use to define a computationally adequate denotational semantics for the simply-typed lambda calculus (**STLC**). To ramp up to a richer programming language, we define **calf**$^\omega$, an extension of **calf*** with *unbounded iteration*, in which we define a computationally adequate semantics for Modernized Algol (**MA**), a dialect of Algol [3].

Cost-aware computational adequacy. In both of the case studies we prove a generalization of the classic, *extensional* Plotkin adequacy that we refer to as *cost-aware computational adequacy*. Roughly, whereas the classic adequacy theorem speaks about the extensional content in both the operational and denotational semantics, cost-aware adequacy relates the operational cost with the denotational cost in addition to the extensional behavior of programs. An important contribution of our work is the fact that ordinary adequacy follows immediately from the cost-aware adequacy theorem, which is an instance of a more general principle of **calf** as a *synthetic language* for mediating the interaction of the cost (intension) and behavior (extension) of programs, a point that we shall expand on in Sections I-A3 and I-A4.

[1] In the literature, the computational adequacy sometimes refer to the conjunction of soundness and adequacy as we have defined here.

979-8-3503-3587-3/23/$31.00 ©2023 IEEE

$$\mathsf{tp}^+ : \mathbf{Jdg} \qquad \mathsf{U} : \mathsf{tp}^\ominus \to \mathsf{tp}^+$$
$$\mathsf{tm}^+ : \mathsf{tp}^+ \to \mathbf{Jdg} \qquad \mathsf{F} : \mathsf{tp}^+ \to \mathsf{tp}^\ominus$$
$$\mathsf{tm}^\ominus(X) \coloneqq \mathsf{tm}^+(\mathsf{U}(X))$$
$$\mathsf{ret} : (A : \mathsf{tp}^+, a : \mathsf{tm}^+(A)) \to \mathsf{tm}^\ominus(\mathsf{F}(A))$$
$$\mathsf{bind} : \{A : \mathsf{tp}^+, X : \mathsf{tp}^\ominus\}\ \mathsf{tm}^\ominus(\mathsf{F}(A)) \to (\mathsf{tm}^+(A) \to \mathsf{tm}^\ominus(X)) \to \mathsf{tm}^\ominus(X)$$
$$\mathsf{step} : \{X : \mathsf{tp}^\ominus\}\ \mathbb{C} \to \mathsf{tm}^\ominus(X) \to \mathsf{tm}^\ominus(X)$$
$$\P_{\mathsf{E}} : \mathbf{Jdg}$$
$$\P_{\mathsf{E}}/\mathsf{uni} : \{u, v : \P_{\mathsf{E}}\}\ u = v$$
$$\bigcirc\mathcal{J} \coloneqq \P_{\mathsf{E}} \to \mathcal{J}$$
$$\mathsf{step}/\P_{\mathsf{E}} : \{X, e, c\}\ \bigcirc(\mathsf{step}^c(e) = e)$$

Fig. 1. A fragment of the signature of **calf**.

Synthetic denotational semantics. The denotational models we define are also synthetic in a more traditional sense: the type structure of the object language is implemented as simple compositions of the corresponding type structures in the metalanguage that do not involve complex analytic constructions typical of classic domain theory.

The benefit of synthetic theories are both theoretical and practical. The axioms of a synthetic theory are useful *abstractions* that reveal the fundamental structures and seal away irrelevant details of the mathematical objects at hand. This has a tangible impact on users of the theory; although a programming languages researcher may not care about how a fixed-point operator is implemented, they will certainly need to use the universal property of the fixed-point to prove theorems about programs. Viewing **calf** as a metalanguage for cost-aware denotational semantics, the synthetic nature of the theory is reflected in both the interpretation of type structures and the treatment of the interaction of intension/extension.

A. *calf: a cost-aware logical framework*

In this section, we recall the key components of **calf** as a type theory and framework for cost analysis; we defer to Niu et al. [2] for more details. We present a fragment of the signature of **calf** in Fig. 1.

1) Dependent call-by-push-value. **calf** is defined as an extension of the dependent call-by-push-value calculus of Pédrot and Tabareau [4]. Recall that the theory of call-by-push-value (CBPV) can be extracted from the Eilenberg-Moore category arising from a monad that encodes the computational effect. More concretely, there are two classes of types in CBPV: the *value/positive* types classifying values, and the *computation/negative* types classifying computations. Semantically, value types correspond to plain sets while computation types correspond to *algebras* for the given monad. The type constructors U, F bridge this stratification of values and computations and corresponds to a free-forgetful adjunction in the semantics. A computation of the type $\mathsf{F}(A)$ is called a *free computation*, and ret and bind are the introduction and elimination forms of the free computations.

2) Cost as a computational effect. As a theory, **calf** is paramterized by an (ordered) monoid $(\mathbb{C}, +, \leq, 0)$. The cost structure of programs is generated from a single computational effect step (see Fig. 1), which one may think operationally as incurring the given cost onto a computation.

As a dependent CBPV calculus, **calf** supports a simple equational theory for reasoning about the cost of computations. For instance, Niu et al. [2] defines an internal predicate $\mathsf{hasCost}_A(e, c) \coloneqq \Sigma a : A.\ e = \mathsf{step}^c(\mathsf{ret}(a))$ that defines when a computation has a given cost.

3) The interaction intension and extension. A key innovation of **calf** as a cost analysis framework is a solution to the problem of *exotic programs*. Traditional accounts of cost structure in type theory employs the cost monad/writer monad $\mathbb{C} \times -$, so that a cost-aware/effectful program of type A is rendered as a term of type $\mathbb{C} \times A$. One thinks of an effectful program in this setting as a program instrumented with a counter that returns the incurred cost. However, this encoding is transparent enough so that the counter is allowed to interfere with the *behavior* of the program; such programs are called *exotic* by Niu et al. [2] because one cannot extract from it an ordinary, cost-unaware program.

Because the free computations and the cost effect step are *abstract*, there is no way to define such exotic programs, which is an internal theorem one may specify and prove in **calf**. Semantically, the free computations may be implemented using the writer monad on an appropriate cost monoid, but it is important that this fact is not exposed in the theory. In order to work with cost effects in the abstract, **calf** introduced a pair of *modalities* for the interaction of intension and extension.

4) Modalities for intension and extension. The first problem one encounters working in a cost-sensitive/intensional setting where the cost effect is abstract is function extensionality. For example, consider the merge sort and insertion sort algorithms. Under the usual cost model, these algorithms are most definitely distinct as far as cost is concerned. However, because they are both sorting algorithms, they are equal in extension/behavior, and by functional extensionality, they are equal! In **calf** this contradiction may be resolved by the following observation: equality of extension/behavior may be analyzed in a special phase called the *extensional phase* in which the cost effect is trivial. Technically, the extensional phase is generated by a distinguished proposition $\P_{\mathsf{E}}$ along with the axiom $\mathsf{step}/\P_{\mathsf{E}}$ (see Fig. 1); whenever we are in a context in which $\P_{\mathsf{E}}$ is derivable, $\mathsf{step}/\P_{\mathsf{E}}$ stipulates that step is trivial, and therefore we require ordinary extensional reasoning. There are no closed proofs of $\P_{\mathsf{E}}$, but no other structures are assumed aside from the fact it is a proposition.

The extensional phase generates a pair of modalities for intension and extension. The extensional modality is defined as $\bigcirc(A) \coloneqq \P_{\mathsf{E}} \to A$, which simply internalizes the derivability of $\P_{\mathsf{E}}$. Given a type A, one can think of $\bigcirc A$ as the extensional part of A; in terms of the cost monad the unit of the extensional modality is the projection map $\mathbb{C} \times A \to A$. Complementary

to the extensional modality is the intensional modality, which is defined as a 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 extensional part of the intensional part is trivial".

In **calf** one can use these modalities to manage the interaction of the intension and extension. For instance, although it is not the case that merge sort and insertion sort are equal in the *empty* context, one can derive their equality in the extensional phase, *i.e.* one has $\bigcirc(\mathsf{mergeSort} = \mathsf{insSort})$. On the other hand, one may use the intensional modality to *seal away* cost structures, which is useful in applications such as program optimization and noninterference.

The phase distinction of intension and extension. The interaction of the intension and extension in **calf** is an instance of a more general phenomenon of *phase distinctions* in the sense of the theory of ML modules; as explained in Sterling and Harper [5], Niu et al. [2], the (non)interaction of intensional structure with the extensional behavior of a cost-aware function is formally identical to the (non)interaction of dynamic components with static components in a module functor. Consequently, one can think of **calf** as a synthetic language for the *phase distinction* of intension and extension, and a **calf** program is said to be **phase-separated** if it exploits the interaction of the intensional and extensional modalities.

B. Cost-aware computational adequacy

These ideas were executed on several case studies in Niu et al. [2], including Euclid's algorithm for the greatest common divisor, amortized analysis batched queues, and sequential and parallel sorting algorithms. An important feature of these analyses is that they all employed their own cost models, which follows the prevailing convention of algorithms research community. Although all the cost models of *op. cit.* are intuitively reasonable, the authors did not provide a formal theory for *why* certain cost models *are* reasonable; however, it was conjectured that this may be achieved via a cost-aware version of Plotkin's adequacy theorem.

We provide a positive answer to this conjecture. For the following, suppose that we have defined inside **calf** a programming language **P** along with an evaluation relation $\Downarrow\colon \mathbf{P} \to \mathbb{N} \to \mathbf{P} \to \mathsf{tp}^+$. A denotational semantics $[\![-]\!]$ of **P** satisfies **cost-aware computational adequacy** when the following holds:

> For all closed programs of base type $\vdash_{\mathbf{P}} e : \mathsf{bool}$, if $[\![e]\!] = \mathsf{step}^c(\mathsf{ret}(b))$ for some $b : \mathsf{bool}$ and $c : \mathbb{N}$, then $e \Downarrow^{\eta_\bullet c}_{\P_{\mathsf{E}}} \overline{b}$.

In the above, bool is the boolean type in **P**, and $\overline{b}$ sends a **calf** boolean b to its numeral in bool. We write $\eta_\bullet : A \to \bullet A$ for the unit of the intensional modality, and the relation $\Downarrow_{\P_{\mathsf{E}}}$ is a *phase-separated* version of the evaluation relation that we will define shortly. Roughly, cost-aware computational adequacy states that if the denotation of a program is equal to a value incurring some cost, then the program must also *compute* to the same value with the same cost.

Phase-separated evaluation. In order to explain phase-separated evaluation, let us consider a statement of cost-aware adequacy using the usual operational cost semantics $\Downarrow$ and observe what goes wrong. Suppose we have a proof of the extensional phase $u : \P_{\mathsf{E}}$, and consider a closed program $e : \mathsf{bool}$ such that $e = \mathsf{step}^c(\mathsf{ret}(\{\,\mathsf{tt}, \mathsf{ff}\,\}))$. We have to show that $e \Downarrow^c \{\,\mathsf{tt}, \mathsf{ff}\,\}$. But because we are in the extensional phase, we also have $\mathsf{step}^c(\mathsf{ret}(\{\,\mathsf{tt}, \mathsf{ff}\,\})) = \mathsf{step}^0(\mathsf{ret}(\{\,\mathsf{tt}, \mathsf{ff}\,\}))$. Therefore, we also have to show that $e \Downarrow^0 \{\,\mathsf{tt}, \mathsf{ff}\,\}$! Now, if $c \neq 0$, we have a contradiction if adequacy holds, because the evaluation relation is deterministic: if $e \Downarrow^c v$ holds, then it holds for unique c and v.

So we would like the relation $\Downarrow$ to *restrict* to a cost-unaware evaluation relation in the extensional phase. This is the purpose of the phase-separated evaluation relation: we define a relation $\Downarrow_{\P_{\mathsf{E}}}\colon \mathbf{P} \to \bullet\mathbb{N} \to \mathbf{P} \to \mathsf{tp}^+$ whose cost component is *sealed* by the intensional modality. We can define such a relation $\Downarrow_{\P_{\mathsf{E}}}$ and prove that it becomes equivalent to the cost-unaware evaluation relation $e \Downarrow v$ in the extensional phase. Consequently, the problem above is resolved because the contradictory evaluation costs are sealed away by the intensional modality and invisible in the extensional phase.

1) Synthetic cost-aware denotational semantics. The contribution of our work is the development of the preceding idea with two concrete programming languages: the simply-typed lambda calculus (**STLC**) and Modernized Algol (**MA**). First, we axiomatize an extension of **calf** with universes and inductive types dubbed **calf***. We define the syntax and operational semantics of **STLC** in **calf***, construct a cost-aware denotational semantics of **STLC**, and prove the model to be computationally adequate in the sense describe above. Next, we axiomatize an extension of **calf*** with unbounded iteration dubbed **calf**$^\omega$ and carry out a similar construction for **MA**, a language with first-order stores and while loops.

In both case studies we will rely on the respective meta-languages to define conceptually simple and *synthetic* models of the object programming languages. Here we emphasize that our models are synthetic in two orthogonal senses. First, as we discussed in Section I-A4, **calf** is a theory for phase-separated constructions. In more geometric language, one can think of **calf** types as families of cost/intensional structures indexed by behavioral/extensional specifications. The benefit of working in a language for such indexed constructions is that one may always *project* the index of the family to obtain the ordinary, extensional content of an object. This is exemplified in our account of cost-aware computational adequacy: the extensional Plotkin-type adequacy theorem **follows immediately as a corollary** of cost-aware computational adequacy, a property that is *not* enjoyed by prior work on denotational semantics in type theory, which we discuss more in Section I-D.

Our work is also synthetic in a more traditional sense: types and computations of the object language is defined via simple constructions using the corresponding structures in the metalanguage. For instance, while loops of **MA** may

be interpreted straightforwardly as an *iteration* primitive of **calf**$^{\omega}$ satisfying the expected computation laws. By isolating the essential properties necessary to develop the computational adequacy proof, we refine and provide a way to interpret classic accounts of adequacy in multiple metatheories.

C. Models of *calf** and *calf*$^{\omega}$

To show that a synthetic theory is sensible, one must exhibit an interpretation or model of the theory in terms of previously understood concepts. Following the work of Niu et al. [2], we prove the consistency of **calf*** and **calf**$^{\omega}$ by means of a model construction. Similar to authors of *op. cit*., we define **calf*** and **calf**$^{\omega}$ as signatures of the *logical framework* of locally cartesian closed categories (lccc's). One can think of the language of this logical framework as an extensional dependent type theory with a universe of judgments closed under dependent products, dependent sums, and extensional equality. A model of a theory (*e.g*. **calf**) associated to a signature is given by an implementation of the constants in the signature in any other lccc.

The authors of *op. cit*. defines a model of **calf** called the *counting model* in an arbitrary presheaf topos equipped with a proposition representing the semantic extensional phase. The counting model is itself an extension of the Eilgenberg-Moore model of CBPV associated to the cost monad. We first extend the counting model of **calf** with an interpretation of inductive types and universes based on the *weaning models* of dependent CBPV [4] to obtain a model of **calf***. We then modify this model to account for partiality, resulting in a model for **calf**$^{\omega}$.

D. Related work

1) Cost-aware denotational semantics. The cost-sensitive generalization of classic extensional denotational semantics has been studied in the context of the formalization of recurrence extraction of higher-order functional programs [6, 7]. One central idea in the work of *op. cit*. is the stratification of the framework across two languages: the programming language and the syntactic recurrence language. The extraction first computes a *syntactic recurrence* from the input program, which is then turned into a semantic recurrence suitable for mathematical manipulation in using a denotational semantics based on *sized domains*. The extraction procedure is shown to be sound, which in conjunction with a Plotkin-type adequacy theorem for the denotational semantics produces a framework for doing algorithm analysis à la recurrence relations.

The work of Kavvos et al. [7] is different from ours in several aspects. First, *op. cit*. defines a denotational semantics for a variant of PCF, while we define the denotational semantics of **MA**, an imperative language with first-order store and while loops (and consequently no mechanism for defining arbitrary fixed-points). Second, the adequacy theorem of *op. cit*. only speaks about cost indirectly through the recurrence extraction, whereas we prove a direct cost-aware adequacy theorem that allows one to directly reason about cost of the source program. Lastly, the most important difference stems from the fact that we work in a formalized metatheory (**calf**) that allows for synthetic constructions as discussed in Sections I-A4 and I-B1. Whereas Kavvos et al. [7] work in a classical set-theoretic metatheory and use classic domain-theoretic constructions, we promote a more abstract approach based on an axiomatization of the necessary domain structures and the interaction of intension and extension. Moreover, because **calf** (and its extensions) is a dependent type theory, one may also use it as programming language, which Niu et al. [2] has shown to be a fruitful endeavor in the context of both the cost analysis and behavioral verification of programs.

2) Denotational semantics of Algol. Algol and Algol-like languages have been widely-studied in terms of both denotational and operational semantics, and we do not recall the specifics of the language here; for more details we defer to the definitive sources on the subject [8, 9, 10]. We present a denotational model for Modernized Algol (**MA**), a version Algol presented in Harper [3]. **MA** features first-order store and unbounded iteration, a call-by-value operational semantics, and a categorical separation of expressions and commands. The denotational semantics we define for **MA** is similar to the presheaf model of Algol [11]; however there are some important differences that we discuss in Section V.

The main improvement of our work is the cost-aware aspect of the denotational semantics and an axiomatization of the properties in **calf**$^{\omega}$ that are necessary to prove computational adequacy. Moreover, as mentioned in Section I-B, our results easily restricts to the classic adequacy theorem for denotational models of **MA** extensionally.

3) Game semantics of Algol. The cost semantics of Algol-like languages has also been studied in the context of *game semantics*. In Ghica [12], the author introduces a denotational model of ICA (Idealized Concurrent Algol) based on an extension of Hyland-Ong-style game semantics that tracks the incurrence of cost. The model is proven to be adequate and in fact fully abstract for the cost-sensitive operational semantics. In contrast to *op. cit*., which studies a concrete model, our work is centered around the axioms needed to define an adequate model of **MA**. Although we validate our axiomatic framework using domain-theoretic constructions, there is no reason that the axioms cannot be satisfied using other models (*e.g*. a game semantics model).

4) Synthetic domain theory. Ever since the pioneering work of Scott on the domain-theoretic semantics for programming languages, there has been much interest [14, 15, 16, 17] in finding set-theoretic universes (in other words, topoi) that embed concrete categories of domains, which would furnish a rich intuitionistic/type-theoretic framework for defining reasoning about domain-theoretic constructions.

Synthetic domain theory (SDT) is an elegant and powerful approach for modeling programming languages, but does not immediately provide a synthetic language for talking about cost-aware computation. A good way to situate our work is to view topoi with an SDT theory as *models* for the kind of metalanguages we promote in this paper. In fact, we hope that by constructing models of **calf** in SDT topoi we can extend our results to Plotkin's PCF, thereby truly generalizing Plotkin's

original adequacy result to a cost-sensitive setting; we shall come back to this point in Section IX. Lastly, whereas the axioms of **calf**$^{\omega}$ may be interpreted in any presheaf topos, the axioms of SDT require considerably more work to model. Therefore it is reasonably to view our work as an intermediate step between the denotational semantics of total computations and general higher-order recursion that is equipped with a relatively simple class of models.

5) Denotational semantics in guarded type theory. More recently, Møgelberg and Paviotti [18], Paviotti et al. [19] promoted the use of *guarded dependent type theory* (gDTT) for doing synthetic denotational semantics. Similar to our approach of using a type-theoretic metalanguage, Møgelberg and Paviotti [18] defines a denotational semantics for FPC in gDTT and prove it to be computationally adequate in the traditional sense. By working in the guarded recursion setting, the denotational semantics of *op. cit*. is also intensional in a certain sense — programs of the guarded lift monad record the number of "steps" they take to yield an answer. However the steps of guarded recursion and the steps in **calf** serve orthogonal purposes: the former is used to ensure that every guarded endofunction has a fixed-point, while the latter is a cost accounting mechanism that is detached from the question of the existence of programs. Consequently, the phase distinction used in **calf** to achieve integration of cost and behavior *cannot* be used in guarded type theory to erase steps.

The lack of an ergonomic theory of the interaction of intension and extension can be seen in the proof of the extensional adequacy theorem of *op. cit*: the adequacy of the *extensional* denotational semantics does *not* follow immediately as a corollary of its intensional counterpart. In fact the extensional denotational semantics requires a separate construction involving the guarded version of the delay monad and the associated weak bisimilarity relation. This work did not disappear in the context of **calf**. By employing a synthetic language of cost-aware constructions one isolates the interaction of intension and extension and the associated principles that are verified once and for all in the model and deployed more generally than the ad-hoc constructions of *op. cit*.

6) Compiler correctness. Lastly, we outline some connections of our work to the area of compiler correctness Patterson and Ahmed [20], Perconti and Ahmed [21], Ahmed [22], Mates et al. [23], Benton and Hur [24], Amadio et al. [25]. In the early days of the mathematization of the study of programming languages, the primary purpose of denotational semantics is to explain the meaning of programs in terms of previously established and undertood mathematical structures. However, as the field and synthetic methods developed, denotational semantics took on more of a logical character: models look more like *translations* between different *languages*, a view point that is expressed in Jung et al. [26]. Consequently, one may view denotational semantics as a sort of "compiler" from the object language into the semantic domain and computational adequacy as a sort of compiler correctness argument. Traditionally, compiler verification is concerned with the *functional* or *extensional* correctness of the compilation process, but there has been a concerted effort to incorporate cost-sensitive properties [25]. In a similar vein, the *cost-aware* metalanguage for synthetic denotational semantics we contribute is a first step towards an ergonomic framework for studying the *intensional* aspects of compilation.

II. CALF*: EXTENDING CALF WITH UNIVERSES AND INDUCTIVE TYPES

In this section we present an extension of **calf** dubbed **calf*** and use it to define and study the **STLC**.

A. Universes

Following Pédrot and Tabareau [4], we axiomatize a *pair* of universes $\mathsf{Univ}^{+}, \mathsf{Univ}^{\ominus}$ classifying value types and computation types respectively. We require that the pair of universes is closed under **calf** types. In this paper we define type families (*i.e*. functions whose codomain is tp^{+}) in a "large elimination" style. This may be unfolded by first defining a family of type *codes* (*i.e*. functions whose codomain is Univ^{+}) and decoding.

B. Inductive types

Because **calf** is an extensional type theory, general inductive types may be encoded by W-types. However, in practice we will use a more natural presentation like the following:

$$\begin{aligned} &\textbf{Inductive}\ \mathbb{N} : \mathsf{tp}^{+}\ \textbf{where} \\ &\quad \mathsf{zero} : \mathbb{N} \\ &\quad \mathsf{suc} : \mathbb{N} \to \mathbb{N} \end{aligned} \tag{1}$$

In general a declaration like Eq. (1) should be thought of defining the *code* of an inductive type. Moreover, because inductive *families* can be defined using *indexed containers* [27], which in turn can be defined using W-types, we will also use a similar notation for defining inductive families.

C. Uniqueness of cost bounds

The theory of cost effects introduced in Niu et al. [2] is sufficient to define and *compose* cost bounds of programs (see Section I-A2). However, in order to prove adequacy, we have to be able to go the other way: it needs to be the case that a given cost bound may be shown to be *unique*:

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

Note that because the premise of $\mathsf{step/inj}$ could have been derived using a proof of the extensional phase, we must seal the equation $c = c'$ by the intensional modality. We will show that $\mathsf{step/inj}$ holds in an extension of the counting model of **calf** in Section VII.

III. WARM-UP: STLC

In this section, we define and study a cost-aware denotational semantics for the STLC. In the following, we suppress some notation from meta-level terms, *i.e*. we write $e : A$ for $e : \mathsf{tm}^{+}(A)$.

A. Representing object languages in *calf**

The exact mechanism by which object-level syntax is defined is immaterial for our purposes; we may choose from a variety of first-order encodings definable using inductive types/families. As an example, we will present an intrinsically-typed nameless representation for STLC based on Benton et al. [28]. We write *e.g.* bool for object-level syntactic phrases.

B. Syntax of the STLC

We consider a version of STLC with a base type of observations $\mathsf{bool} : \mathsf{Ty}$ with two points $\mathsf{tt}, \mathsf{ff} : \mathsf{bool}$ and the function type $\Rightarrow : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty}$. Because we work with an intrinsic encoding, the type of terms is indexed by an object-level context $\mathsf{Con} \coloneqq \mathsf{list}(\mathsf{Ty})$ and type:

$$\begin{aligned}
&\textbf{Inductive}\ \mathsf{Tm} : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{tp}^+\ \textbf{where}\\
&\quad \mathsf{var} : \{\Gamma, A\}\ \mathsf{Var}(\Gamma, A) \to \mathsf{Tm}(\Gamma, A)\\
&\quad \mathsf{lam} : \{\Gamma, A_1, A_2\}\ \mathsf{Tm}(A_1 :: \Gamma, A_2) \to\\
&\qquad \mathsf{Tm}(\Gamma, A_1 \Rightarrow A_2)\\
&\quad \mathsf{ap} : \{\Gamma, A_1, A_2\}\ \mathsf{Tm}(\Gamma, A_1 \Rightarrow A_2) \to\\
&\qquad \mathsf{Tm}(\Gamma, A_1) \to \mathsf{Tm}(\Gamma, A_2)\\
&\quad \mathsf{tt}, \mathsf{ff} : \{\Gamma\}\ \mathsf{Tm}(\Gamma, \mathsf{bool})
\end{aligned}$$

In the above, elements of the family Var represents proofs for variable indexing:

$$\begin{aligned}
&\textbf{Inductive}\ \mathsf{Var} : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{tp}^+\ \textbf{where}\\
&\quad \mathsf{here} : \{\Gamma, A\}\ \mathsf{Var}(A :: \Gamma, A)\\
&\quad \mathsf{next} : \{\Gamma, A, A_1\}\ \mathsf{Var}(\Gamma, A_1) \to \mathsf{Var}(A :: \Gamma, A_1)
\end{aligned}$$

Substitution. We write a substitution from Γ to Γ' as $\mathsf{Sub}(\Gamma, \Gamma')$. Given a substitution $\sigma : \mathsf{Sub}(\Gamma, \Gamma')$ and term $e : \mathsf{Tm}(\Gamma, A)$, we write $e[\sigma] : \mathsf{Tm}(\Gamma', A)$ for the result of the substitution. We not dwell on the all the expected properties of substitution and assume that they hold as usual. For instance we assume the following, which is often used in the proof of the fundamental theorem of a logical relation:

Proposition III.1. *Given* $e : \mathsf{Tm}(A :: \Gamma, A')$, $\sigma : \mathsf{Sub}(\Gamma, \mathsf{nil})$, *and* $e' : \mathsf{Tm}(\mathsf{nil}, A)$, *we have that* $e[\uparrow^A \sigma][e'] = e[\mathsf{cons}(e', \sigma)]$.

In the above, $\mathsf{cons}(e, \sigma) : \mathsf{Sub}(A :: \Gamma, \Gamma')$ denotes a substition extended by a term $e : \mathsf{Tm}(\Gamma', A)$, and $\uparrow^A \sigma : \mathsf{Sub}(A :: \Gamma, A :: \Gamma')$ denotes weakening.

C. Operational semantics

We work with a call-by-value operational semantics for STLC, which may be encoded using the following type families:

$$\begin{aligned}
&\textbf{Inductive}\ \mathsf{Val} : \{\Gamma, A\}\ \mathsf{Tm}(\Gamma, A) \to \mathsf{tp}^+\ \textbf{where}\\
&\textbf{Inductive}\ \mapsto : \{A\}\ \mathsf{Pg}(A) \to \mathsf{Pg}(A) \to \mathsf{tp}^+\ \textbf{where}
\end{aligned}$$

In the above, we write $\mathsf{Pg}(A) \coloneqq \mathsf{Tm}([], A)$ for the type of closed STLC terms. Evaluation may be defined as usual $e \Downarrow v \coloneqq e \mapsto^* v \times \mathsf{Val}(v)$. In a similar fashion, we may define the cost-aware evaluation relation by using a $\mathbb{N}$-indexed version of the reflexive-transitive closure of one-step transition $e \Downarrow^c v \coloneqq e \mapsto^{(c)} v \times \mathsf{Val}(v)$.

Notation. We write Val for the type of values of **STLC**: $\mathsf{Val}(\Gamma, A) = \Sigma e : \mathsf{Tm}(\Gamma, A).\ \mathsf{Val}(e)$. We write $\mathsf{Sub}^{\mathsf{v}}(\Gamma, \Gamma')$ for a substitution whose terms are all values.

1) Phase-separated cost semantics. As discussed in Section I-B, we cannot directly use the cost-aware evaluation relation defined above in the statement of the cost-aware adequacy theorem. Instead, we define a more refined version of the cost-aware evaluation relation that *restricts* to the ordinary evaluation relation in the extensional phase. To this end, we may define a *phase-separated* version of the cost-aware reflexive transitive closure:

$$\begin{aligned}
&\textbf{Inductive}\ \mapsto_{¶_{\mathsf{E}}} : \{A\}\ \mathsf{Pg}(A) \to \bullet\mathbb{N} \to \mathsf{Pg}(A) \to \mathsf{tp}^+\ \textbf{where}\\
&\quad \mathsf{refl} : \{e\}\ e \mapsto^{\eta \bullet^0}_{¶_{\mathsf{E}}} e\\
&\quad \mathsf{trans} : \{c, e\}\ e \mapsto e_1 \to e_1 \mapsto^{c}_{¶_{\mathsf{E}}} e_2 \to e \mapsto^{c(\bullet+)\eta\bullet^1}_{¶_{\mathsf{E}}} e_2
\end{aligned}$$

Cruically, this relation becomes equivalent to the ordinary reflexive transitive closure under the extensional phase:

Proposition III.2. *Given* $u : ¶_{\mathsf{E}}$ *and* $c : \mathbb{N}$, *we have that* $e \mapsto^{\eta\bullet^c}_{¶_{\mathsf{E}}} v$ *if and only if* $e \mapsto^* v$.

Consequently, we may define phase-separated evaluation as $e \Downarrow^c_{¶_{\mathsf{E}}} v \coloneqq e \mapsto^{(c)} v \times \mathsf{Val}(v)$, which satisfies a similar restriction property:

Proposition III.3. *Given* $u : ¶_{\mathsf{E}}$, *we have that* $e \Downarrow^{\eta\bullet^c}_{¶_{\mathsf{E}}} v$ *if and only if* $e \Downarrow v$ *for all* $c : \mathbb{N}$.

D. A cost-aware denotational semantics for STLC

We now define a denotational semantics for STLC in **calf*** based on the standard polarized decomposition of call-by-value. Types are interpreted as follows:

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

The interpretation for types is extended to contexts in the obvious way, and the interpretation of variables is given by projection. For the interpretation of terms, we insert cost effects for elimination forms to account for steps in the operational semantics:

$$\begin{aligned}
[\![-]\!]_{\mathsf{Tm}} &: \{\Gamma, A\}\ \mathsf{Tm}(\Gamma, A) \to [\![\Gamma]\!]_{\mathsf{Con}} \to \mathsf{F}([\![A]\!]_{\mathsf{Ty}})\\
[\![\mathsf{var}(v)]\!]_{\mathsf{Tm}} &= \mathsf{ret} \circ [\![v]\!]_{\mathsf{Var}}\\
[\![\mathsf{tt}]\!]_{\mathsf{Tm}} &= \lambda_.\ \mathsf{ret}(\mathsf{tt})\\
[\![\mathsf{ff}]\!]_{\mathsf{Tm}} &= \lambda_.\ \mathsf{ret}(\mathsf{ff})\\
[\![\mathsf{lam}(e)]\!]_{\mathsf{Tm}} &= \lambda\gamma : [\![\Gamma]\!]_{\mathsf{Con}}.\ \mathsf{ret}(\lambda a : [\![A_1]\!]_{\mathsf{Ty}}.\ [\![e]\!]_{\mathsf{Tm}}(a, \gamma))\\
[\![\mathsf{ap}(e, e_1)]\!]_{\mathsf{Tm}} &= \lambda\gamma : [\![\Gamma]\!]_{\mathsf{Con}}.\ f \leftarrow [\![e]\!]_{\mathsf{Tm}}(\gamma); a \leftarrow [\![e_1]\!]_{\mathsf{Tm}}(\gamma);\\
&\qquad \mathsf{step}(f(a))
\end{aligned}$$

To state the soundness theorem, it will be useful to isolate part of the model dealing with values: $[\![-]\!]_{\mathsf{Val}} : \{\Gamma, A\}\ \mathsf{Val}(\Gamma, A) \to [\![\Gamma]\!]_{\mathsf{Con}} \to [\![A]\!]_{\mathsf{Ty}}$. A substitution also induces an action on contexts, which we denote by $[\![-]\!]_{\mathsf{Sub}} : \{\Gamma, \Delta\}\ \mathsf{Sub}^{\mathsf{v}}(\Gamma, \Delta) \to [\![\Delta]\!]_{\mathsf{Con}} \to [\![\Gamma]\!]_{\mathsf{Con}}$.

E. Soundness

First, we show that operational cost bounds are also denotational cost bounds. The following lemma states that the denotational semantics commutes with substitution of *values*:

Lemma III.4 (Compositionality)**.** *Given* $\mathsf{e} : \mathsf{Tm}(\Gamma, A)$ *and* $\gamma : \mathsf{Sub}^{\mathsf{v}}(\Gamma, \Delta)$*, we have that* $[\![\mathsf{e}]\!]_{\mathsf{Tm}}([\![\gamma]\!]_{\mathsf{Sub}}(\delta)) = [\![\mathsf{e}[\gamma]]\!]_{\mathsf{Tm}}(\delta)$ *for all* $\delta : [\![\Delta]\!]_{\mathsf{Con}}$.

Because we will be stating the soundness theorem in terms of the phase-separated evaluation, we first define a version of the cost effect for the *sealed* cost structure:

$$\begin{aligned}
&\mathsf{step}^{-}_{\P_{\mathsf{E}}}(-) : \{X\}\ \bullet\mathbb{C} \to \mathsf{tm}^{\ominus}(X) \to \mathsf{tm}^{\ominus}(X)\\
&\mathsf{step}^{\eta\bullet c}_{\P_{\mathsf{E}}}(e) = \mathsf{step}^{c}(e)\\
&\mathsf{step}^{*(u)}_{\P_{\mathsf{E}}}(e) = e
\end{aligned}$$

It is easy to see that $\mathsf{step}_{\P_{\mathsf{E}}}$ satisfy laws analogous to step.

Theorem III.5 (Soundness)**.** *Given* $\mathsf{e}, \mathsf{v} : \mathsf{Pg}(A)$*, if* $\mathsf{e} \Downarrow^{c}_{\P_{\mathsf{E}}} \mathsf{v}$*, then there is a term* $p : \mathsf{Val}(\mathsf{v})$ *and* $[\![\mathsf{e}]\!]_{\mathsf{Tm}} = \mathsf{step}^{c}_{\P_{\mathsf{E}}}(\mathsf{ret}([\![\mathsf{v}, p]\!]_{\mathsf{Val}}))$.

Corollary. *Given* $\mathsf{e}, \mathsf{v} : \mathsf{Pg}(A)$*, if* $\mathsf{e} \Downarrow^{\eta\bullet c}_{\P_{\mathsf{E}}} \mathsf{v}$*, then there is a term* $p : \mathsf{Val}(\mathsf{v})$ *and* $[\![\mathsf{e}]\!]_{\mathsf{Tm}} = \mathsf{step}^{c}(\mathsf{ret}([\![\mathsf{v}, p]\!]_{\mathsf{Val}}))$.

F. Computational adequacy

1) Logical relation for adequacy. Following the classic adequacy proof of Plotkin, we prove our cost-aware adequacy theorem by means of a logical relations construction. First, we define a binary logical relation relating the values of STLC of type $A : \mathsf{Ty}$ with values in the semantic domain $[\![A]\!]^{\mathbf{STLC}}_{\mathsf{Ty}}$ by induction on A:

$$\begin{aligned}
&\approx : \{A\}\ \mathsf{Pg}(A) \to [\![A]\!]_{\mathsf{Ty}} \to \mathsf{tp}^{+}\\
&\mathsf{b} \approx_{\mathsf{bool}} b = (\mathsf{b} = \overline{b})\\
&\mathsf{e} \approx_{A_1 \Rightarrow A_2} e = (\Sigma \mathsf{e}_2 : \mathsf{Tm}(A_1, A_2).\ \mathsf{e} = \mathsf{lam}(\mathsf{e}_2)\\
&\quad \times \mathsf{U}(((\mathsf{e}_1 : \mathsf{Pg}(A_1), e_1 : [\![A_1]\!]_{\mathsf{Ty}}) \to\\
&\quad \mathsf{e}_1 \approx_{A_1} e_1 \to \mathsf{e}_2[\mathsf{e}_1] \approx^{\Downarrow}_{A_2} e(e_1))))
\end{aligned}$$

In the above $\overline{-}$ sends tt to tt and ff to ff, and $-^{\Downarrow}$ lifts a relation on values to computations using the phase-separated evaluation relation defined in Section III-C1:

$$\begin{aligned}
&-^{\Downarrow} : \{A\}\ (\mathsf{Pg}(A) \to [\![A]\!]_{\mathsf{Ty}} \to \mathsf{tp}^{+}) \to\\
&\quad (\mathsf{Pg}(A) \to \mathsf{F}([\![A]\!]_{\mathsf{Ty}}) \to \mathsf{tp}^{+})\\
&\mathsf{e}\ R^{\Downarrow}\ e = \Sigma \mathsf{v} : \mathsf{Pg}(A).\ \Sigma v : [\![A]\!]_{\mathsf{Ty}}.\\
&\quad (\mathsf{e} \Downarrow^{\eta\bullet c}_{\P_{\mathsf{E}}} \mathsf{v}) \times e = \mathsf{step}^{c}(\mathsf{ret}(v)) \times \mathsf{v}\ R\ v
\end{aligned}$$

The relation is readily lifted to contexts:

$$\begin{aligned}
&\textbf{Inductive}\ \approx : \{\Gamma\}\ \mathsf{Inst}(\Gamma) \to [\![\Gamma]\!]_{\mathsf{Con}} \to \mathsf{tp}^{+}\ \textbf{where}\\
&\quad \mathsf{emp} : \mathsf{nil} \approx_{\cdot} \star\\
&\quad \mathsf{cons} : \{\Gamma, \gamma, \gamma, A, \mathsf{a}, a\}\ \mathsf{a} \approx_{A} a \to \gamma \approx_{\Gamma} \gamma \to\\
&\qquad \mathsf{cons}(\mathsf{a}, \gamma) \approx_{A::\Gamma} (a, \gamma)
\end{aligned}$$

2) Fundamental theorem. As usual, we may prove the fundamental theorem of the logical relation by induction on terms:

Theorem III.6 (FTLR)**.** *Given a STLC term* $e : \mathsf{Tm}(\Gamma, A)$*, if* $\gamma \approx_{\Gamma} \gamma$*, then* $e[\gamma] \approx^{\Downarrow}_{A} [\![e]\!]_{\mathsf{Tm}}(\gamma)$.

Corollary (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* $e \Downarrow^{\eta\bullet c} \overline{b}$.

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

Proof. Because $e = \mathsf{ret}(b) = \mathsf{step}^{0}(\mathsf{ret}(b))$, we may apply Section III-F2 to obtain $e \Downarrow^{\eta\bullet 0}_{\P_{\mathsf{E}}} \overline{b}$. But since we have $u : \P_{\mathsf{E}}$, we have $e \Downarrow \overline{b}$ by Proposition III.3. □

IV. CALF$^{\omega}$: UNBOUNDED ITERATION

In order to define a model of Modernized Algol (see Section V), we will need to extend **calf*** with facilities for modeling while loops. In this section, we present **calf**$^{\omega}$, a metalanguage for *unbounded iteration*. Intuitively, given a "one step" computation $f : A \to B + A$, we axiomatize a computation $\mathsf{iter}(f) : A \to B$ that iterates f until a $\mathsf{inl}(b) : B + A$ is reached. Moreover, we have to account for the fact that both $\mathsf{iter}(f, a)$ and f itself may diverge by using *lifted computations*.

Lifted computations. Although it might be tempting to combine the computational effects of *cost* and *partiality*, doing so will complicate our model construction for **calf**$^{\omega}$. In particular, it is not immediately clear how to assign meaning to a potential divergent *type* computation in the usual adjunction models of CBPV we consider in this paper. Fortunately, we may sidestep this problem by axiomatizing a more general class of *lifted computations* $\mathsf{L}(A)$ that supports possibly divergent computations and isolate among these the previously cost-sensitive (but total) computations $FA \hookrightarrow LA$:

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

Similar to free computations, we may sequence lifted computations, which we notate by $\mathsf{bind}_{\mathsf{L}}$. Note that in contrast to bind, one may only use $\mathsf{bind}_{\mathsf{L}}$ to sequence a lifted computation with another lifted computation. Lifted sequencing satisfies the expected equational laws with respect to the unit of lifting, defined as $\mathsf{ret}_{\mathsf{L}} := \mathsf{lift} \circ \mathsf{ret}$. Moreover, the lifting of free computations commutes with sequencing and the cost effect:

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

We write $a \leftarrow_{\mathsf{L}} e; f(a)$ for $\mathsf{bind}_{\mathsf{L}}(e; f)$.

A. Axioms for iteration

Equipped with this intuition, we may axiomatize iteration in **calf**$^{\omega}$ as follows; note that iteration is only available for lifted computations:

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

As we will see in Section V-F, we need iterative computations to satisfy a certain compactness property, in the sense that whenever an iterative computation $\mathsf{iter}(f, a)$ has a cost bound, there is a finite prefix $\mathsf{seq}(f, k, a)$ that suffices for obtaining that cost bound:

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

In the above we write $\|A\|$ for the propositional truncation of a given type A. Similar to the uniqueness axioms we introduced in Section III, we require that cost bounds for lifted computations are unique:

$$\begin{aligned}\mathsf{step}_{\mathsf{L}}/\mathsf{inj} &: \{A, (a, a' : A)(c, c' : \mathbb{C})\} \\ &\quad \mathsf{step}^{c}(\mathsf{ret}_{\mathsf{L}}(a)) = \mathsf{step}^{c'}(\mathsf{ret}_{\mathsf{L}}(a')) \to \\ &\quad (a = a') \times \bullet(c = c')\end{aligned}$$

we will also postulate that cost bounds on lifted computations may be decomposed:

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

V. Modernized Algol

We define and study a denotational semantics for a variant of Modernized Algol (**MA**) as formulated in Harper [3]. **MA** is a procedural programming language with first-order store and unbounded iteration and obeys a stack discipline in the sense that store assignables are deallocated when they go out of scope. A characteristic feature of **MA** is the distinction between expressions and commmands that reflects the separation of mathematical and effectful computation. Unlike *op. cit.*, we restrict our expression language to a total language (a mild extension of **STLC** as presented in Section III) and extend the command language with a primitive iteration command.

Inductive Ty **where**

Inductive pos : Ty $\to$ tp^{+} **where**

$$\begin{aligned}&\mathsf{unit}, \mathsf{bool}, \mathsf{nat} : \mathsf{Ty} && \mathsf{unit}/\mathsf{pos} : \mathsf{pos}(\mathsf{unit}) \\ &\Rightarrow : \mathsf{Ty} \to \mathsf{Ty} && \mathsf{bool}/\mathsf{pos} : \mathsf{pos}(\mathsf{bool}) \\ &\mathsf{cmd} : \mathsf{Ty} \to \mathsf{Ty} && \mathsf{nat}/\mathsf{pos} : \mathsf{pos}(\mathsf{nat})\end{aligned}$$

Fig. 2. Left: types of **MA**; right: strictly positive types.

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

$$\frac{\Sigma[n] = (\mathsf{bool}, -) \qquad \Gamma \vdash_{\Sigma} m \div \mathsf{unit}}{\Gamma \vdash_{\Sigma} \mathsf{while}[n](m) \div \mathsf{unit}}$$

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

Fig. 3. Selected expressions and commands of **MA**.

The version of **MA** that we present may be some ways away from a realistic programming language, but the methods we develop here are general and may be extended to include richer data structures so that one may program the algorithms studied in Niu et al. [2].

A. Syntax of *MA*

MA is equipped with a small class of inductive types, functions types, and a type $\mathsf{cmd}(A)$ of reified commands that compute expressions of type A (see Fig. 2). We also outline a class of *strictly positive types* that may be used as assignables, which restricts **MA** to *first-order stores*. We define the type of strictly positive types as $\mathsf{Pos} \coloneqq \Sigma A : \mathsf{Ty}.\ \mathsf{pos}(A)$. In order to faciliate readbility, we will not write the proof of strict positivity, *i.e.* we write $\mathsf{bool} : \mathsf{Pos}$ for $(\mathsf{bool}, \mathsf{bool}/\mathsf{pos}) : \mathsf{Pos}$.

As for the **STLC**, we work with an instrinsic encoding of **MA**; the terms of **MA** are presented in Fig. 3. Both the judgment for well-typed commands Cmd and expressions Tm are indexed by a *signature* $\mathsf{Sig} \coloneqq \mathsf{list}(\mathsf{Pos})$ of strictly positive types. Well-typed expressions $e : \mathsf{Tm}(\Sigma, \Gamma, A)$ are written as $\Gamma \vdash_{\Sigma} e : A$ and well-typed commands $m : \mathsf{Cmd}(\Sigma, \Gamma, A)$ are written as $\Gamma \vdash_{\Sigma} m \div A$; note that we make use of a mutual inductive definition because well-typed expressions and commands must be defined simultaneously.

B. Preorder of signatures

In the possible worlds semantics/Kripke models of mutable store, the interpretation of signature-indexed commands and expressions can be thought of as a family of models linked by a contravariant action on the *preorder relation* on the signatures, which represents the stability of the interpretation with respect

to allocation of new assignables. We may define the preorder on signatures as follows:

$$\begin{aligned}&\textbf{Inductive}\ \ \geq : \mathsf{Con} \to \mathsf{Con} \to \mathsf{tp}^+\ \textbf{where}\\ &\quad \mathsf{refl} : \{\Sigma\}\ \Sigma \geq \Sigma\\ &\quad \mathsf{mono} : \{\Sigma, \Sigma', A\}\ \Sigma' \geq \Sigma \to A :: \Sigma' \geq A :: \Sigma\\ &\quad \mathsf{extend} : \{\Sigma, \Sigma', A\}\ \Sigma' \geq \Sigma \to A :: \Sigma' \geq \Sigma\end{aligned}$$

In other words, we have a proof of $\Sigma' \geq \Sigma$ whenever Σ occurs as a *subsequence* of Σ'.

Proposition V.1. *The relation $\geq$ is reflexive and transitive. We write* $\mathsf{tr} : \{\Sigma'', \Sigma', \Sigma : \mathsf{Sig}\}\ \Sigma'' \geq \Sigma' \to \Sigma' \geq \Sigma \to \Sigma'' \geq \Sigma$ *for the proof of transitivity.*

Given a signature Σ, we write a substitution from Γ to Γ' as $\mathsf{Sub}_\Sigma(\Gamma, \Gamma')$. As for the **STLC** we use $-[-]$ to denote application of a subsititution.

Action of the preorder of signatures. Every future world induces an action on terms of **MA**. Given a proof $p : \Sigma' \geq \Sigma$, we write $\Uparrow^p$ for the induced action on expressions, commands, and substitutions.

C. Operational semantics of *MA*

The operational semantics of **MA** is defined separately for expressions and commands. Expressions execute via substitution as in **STLC**:

$$\begin{aligned}\mathsf{val} &: \{\Sigma, \Gamma, A\}\ \mathsf{Tm}(\Sigma, \Gamma, A) \to \mathsf{tp}^+\\ \mapsto &: \{\Sigma, A\}\ \mathsf{Pg}(\Sigma, A) \to \mathsf{Pg}(\Sigma, A) \to \mathsf{tp}^+\end{aligned}$$

In the above, we defined closed expressions as $\mathsf{Pg}(A) := \mathsf{Tm}(\Sigma, \mathsf{nil}, A)$. Define $\mathsf{Val}(\Sigma, A) := \Sigma e : \mathsf{Pg}(\Sigma, A).\ \mathsf{val}(e)$. In contrast to expressions, commands execute in conjunction with a *store* that contains values associated to the declared assignables. More explicitly, we may define a store as the following family indexed in a signature:

$$\begin{aligned}\mathsf{store} &: \mathsf{Sig} \to \mathsf{tp}^+\\ \mathsf{emp} &: \mathsf{Store}(\mathsf{nil})\\ \mathsf{extend} &: \{\Sigma, A\}\ (p : \mathsf{pos}(A)) \to \mathsf{Val}(\mathsf{nil}, A) \to\\ &\quad \mathsf{Store}(\Sigma) \to \mathsf{Store}((A, p) :: \Sigma)\end{aligned}$$

A *state* is composed of a store and command, defined as $\mathsf{State}(\Sigma, A) := \mathsf{Store}(\Sigma) \times \mathsf{Cmd}(\Sigma, A)$, and we have the following judgments governing the dynamics of states:

$$\begin{aligned}\mathsf{final} &: \{\Sigma, A\}\ \mathsf{State}(\Sigma, A) \to \mathsf{tp}^+\\ \Mapsto &: \{\Sigma, A\}\ \mathsf{State}(\Sigma, A) \to \mathsf{State}(\Sigma, A) \to \mathsf{tp}^+\end{aligned}$$

As an example, we give the rules for while in Section V-C. For brevity we have suppressed the definitions of the other commands; the complete definition may be found in Harper [3]. As usual we define evaluation of expressions as $e \Downarrow_{\mathsf{exp}} v := e \mapsto^* v \times \mathsf{val}(v)$ and evaluation of commands as $(\mu, m) \Downarrow_{\mathsf{cmd}} (\mu', m') := (\mu, m) \Mapsto^* (\mu', m') \times \mathsf{final}(m')$.

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

$$\frac{\mu[n] = \mathsf{tt}}{(\mu, \mathsf{while}[n](m)) \Mapsto (\mu, \mathsf{bnd}(\mathsf{cmd}(m), \mathsf{wk}(\mathsf{while}[n](m))))}$$

Fig. 4. Selected rules for the one-step transition of commands of **MA**; here wk denotes weakening of a term.

$$\begin{aligned}[\![-]\!]^{\mathbf{MA}}_{\mathsf{Ty}^+} &: \mathsf{Pos} \to \mathsf{tp}^+\\ [\![(\mathsf{unit}, -)]\!]^{\mathbf{MA}}_{\mathsf{Ty}^+} &= 1\\ [\![(\mathsf{bool}, -)]\!]^{\mathbf{MA}}_{\mathsf{Ty}^+} &= \mathsf{bool}\\ [\![(\mathsf{nat}, -)]\!]^{\mathbf{MA}}_{\mathsf{Ty}^+} &= \mathbb{N}\end{aligned}$$

$$\begin{aligned}[\![-]\!]^{\mathbf{MA}}_{\mathsf{Sig}} &: \mathsf{Sig} \to \mathsf{tp}^+\\ [\![\cdot]\!]^{\mathbf{MA}}_{\mathsf{Sig}} &= 1\\ [\![A :: \Sigma]\!]^{\mathbf{MA}}_{\mathsf{Sig}} &= [\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}^+} \times [\![\Sigma]\!]^{\mathbf{MA}}_{\mathsf{Sig}}\end{aligned}$$

Fig. 5. Top: the interpretation of strictly positive types; bottom: interpretation of signatures;

1) Phase-separated operational semantics. Similar to the case for **STLC**, in order to state and prove adequacy, we will need a phase-separated version of the transition relation for both expressions and commands where the evaluation cost is *sealed* by the closed modality:

$$\begin{aligned}\mapsto_{\P_{\mathsf{E}}} &: \{\Sigma, A\}\ \mathsf{Pg}(\Sigma, A) \to \bullet\mathbb{N} \to \mathsf{Pg}(\Sigma, A) \to \mathsf{tp}^+\\ \Mapsto_{\P_{\mathsf{E}}} &: \{\Sigma, A\}\ \mathsf{State}(\Sigma, A) \to \bullet\mathbb{N} \to \mathsf{State}(\Sigma, A) \to \mathsf{tp}^+\end{aligned}$$

Using these phase-separated relations we may define phase-separated evaluation for both expressions and commands as $e \Downarrow^c_{\P_{\mathsf{E}}} v := e \mapsto^c_{\P_{\mathsf{E}}} v \times \mathsf{val}(v)$ and $(\mu, m) \Downarrow^c_{\P_{\mathsf{E}}/\mathsf{cmd}} (\mu', m') := (\mu, m) \Mapsto^c_{\P_{\mathsf{E}}} (\mu', m') \times \mathsf{final}(m')$ respectively. As before, phase-separated evaluation restricts to ordinary evaluation in the extensional phase.

D. A denotational model for *MA*

As mentioned in Section V-B, our denotational semantics of **MA** is based on a possible-worlds model of allocation. Consequently, we interepret (closed) commands as *families* of functions that may be executed on any future semantic store (according the the preorder of signatures). Because **MA** only admits first-order store, we may bootstrap the definition by first defining the meaning of strictly positive types, which is independent of the interpretation of signatures (see Fig. 5). The definition of types (strictly positive types omitted) is then allowed to reference the meaning of signatures:

$$\begin{aligned}[\![-]\!]^{\mathbf{MA}}_{\mathsf{Ty}} &: \mathsf{Ty} \to \mathsf{Sig} \to \mathsf{tp}^+\\ [\![A_1 \Rightarrow A_2]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma) &= \mathsf{U}((\Sigma' : \mathsf{Con}) \to \Sigma' \geq \Sigma \to\\ &\qquad [\![A_1]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma') \to \mathsf{F}([\![A_2]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma')))\\ [\![\mathsf{cmd}(A)]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma) &= \mathsf{U}((\Sigma' : \mathsf{Con}) \to \Sigma' \geq \Sigma \to\\ &\qquad [\![\Sigma']\!]^{\mathbf{MA}}_{\mathsf{Sig}} \to \mathsf{L}([\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma') \times [\![\Sigma']\!]^{\mathbf{MA}}_{\mathsf{Sig}}))\end{aligned}$$

The interpretation of types evidently has the structure of a presheaf on signatures. Given $p : \Sigma' \geq \Sigma$, we write $\mathsf{up}(p, a) : [\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma')$ for the action of the restriction on $a : [\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma)$.

1) The model. In Fig. 6 we present the denotational semantics of expression and commands of **MA**. The expression-level denotational semantics for **MA** is defined in a similar style to Section III; reified commands are defined using the mutually recursive interpretation of commands. Commands are defined using a possible-worlds semantics of first-order store. Observe that for any extension $\Sigma' \geq \Sigma$, the meaning of a command of type A are families of *lifted* transformations $[\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma') \to \mathsf{L}([\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma') \times [\![\Sigma']\!]^{\mathbf{MA}}_{\mathsf{Sig}})$. In contrast, because expressions are total and cannot modify the store, they are simply interpreted as *free* computations of the given type. As in the case for the **STLC**, we must insert step's in both the interpretation of expressions and commands when β-reductions occur in the operational semantics.

Here, we highlight the the fact that the while loop of **MA** is interpreted as a "while loop" in **calf**$^\omega$, a feature typical of synthetic denotational semantics. As we shall see in Section V-F, this synthetic interpretation allows for a somewhat involved but elementary proof of computational adequacy.

To prove soundness it will also be useful to restrict the model to values:

$$[\![-]\!]^{\mathbf{MA}}_{\mathsf{Val}} : \{\Sigma, \Gamma, A\}\ \mathsf{Val}(\Sigma, \Gamma, A) \to (\Sigma' : \mathsf{Sig}) \to \Sigma' \geq \Sigma \to [\![\Gamma]\!]^{\mathbf{MA}}_{\mathsf{Con}}(\Sigma') \to [\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma')$$

We may then define the meaning of stores $[\![-]\!]_{\mathsf{St}} : \{\Sigma\}\ \mathsf{Store}(\Sigma) \to [\![\Sigma]\!]^{\mathbf{MA}}_{\mathsf{Sig}}$ using the interpretation values.

Comparison to presheaf models of Algol. A factor that prevents a direct comparison between our model of **MA** and presheaf models of Algol is the treatment of expressions and commands. While **MA** adheres to a strict separation of expressions and commands, Algol in the Reynolds school admit "expressions" that engender effects, which both simplifes and complicates the semantics: because commands need not return values, they may be interpreted using the internal language of the presheaf category, whereas it is not clear how one can do this for the commands of **MA**.

Nonetheless, types in the model of **MA** we define contain the "raw data" specifying a presheaf on the category of signatures. Although we do not emphasize this fact in this paper, we believe it would be possible to prove that the interpretation in fact satisfies the associated naturality conditions.

E. Soundness

Similar to the **STLC**, we may prove a theorem that validates every operational bound as a denotational one.

Theorem V.2 (Soundness)**.** *Given* $e, v : \mathsf{Tm}(\Sigma, A)$*, if* $e \Downarrow^c_{\P_{\mathsf{E}}} v$*, then* $[\![e]\!]^{\boldsymbol{MA}}_{\mathsf{Exp}}(\Sigma, \mathsf{refl}) = \mathsf{step}^c_{\P_{\mathsf{E}}}(\mathsf{ret}([\![v, h]\!]^{\boldsymbol{MA}}_{\mathsf{Val}})(\Sigma, \mathsf{refl}))$ *for some* $h : \mathsf{val}(\Sigma, v)$*. Moreover, given* $m : \mathsf{Cmd}(\Sigma, A)$ *and* $v : \mathsf{Tm}(\Sigma, A)$*, if* $(\mu, m) \Downarrow^c_{\P_{\mathsf{E}}/\mathsf{cmd}} (\mu', \mathsf{ret}(v))$*, then* $[\![m]\!]^{\boldsymbol{MA}}_{\mathsf{Cmd}}(A :: \Sigma, \mathsf{refl}, [\![\mu]\!]_{\mathsf{St}}) = \mathsf{step}^c_{\P_{\mathsf{E}}}(\mathsf{ret}_{\mathsf{L}}([\![v, h]\!]^{\boldsymbol{MA}}_{\mathsf{Val}}(\Sigma, \mathsf{refl}), [\![\mu']\!]_{\mathsf{St}}))$ *for some* $h : \mathsf{val}(\Sigma, v)$*.*

F. Computational adequacy

In this section we prove that the denotational semantics of **MA** defined in Section V-D satisfies the cost-aware computational adequacy theorem from Section I-B with respect to its phase-separated operational semantics. As in the case of the **STLC**, we employ the method of logical relations to prove this result. As foreshadowed by the construction of the model in Section V-D, we have to stage the definition of the logical relation by first defining the relation for strictly positive types in Section V-F1, using this relation to define the *pre*logical relation for commands in Section V-F2, using this to define the logical relation for expressions in Section V-F3, and finally closing the loop by defining the logical relation for commands in Section V-F4. We prove the fundamental theorem of the logical relation in Section V-F5.

1) Logical relation for strictly positive types. For strictly positive types, we relate the values in the semantic domain with their numerals in **MA**:

$$\begin{aligned}
&\textbf{Inductive}\ \approx : \{A : \mathsf{Pos}\}\ \mathsf{Pg}(\cdot, A) \to [\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}^+} \to \mathsf{tp}^+\ \textbf{where}\\
&\quad \mathsf{unit}/\mathsf{base} : \star \approx_{\mathsf{unit}} \star\\
&\quad \mathsf{bool}/\mathsf{base} : (b : \mathsf{bool}) \to \overline{b} \approx_{\mathsf{bool}} b\\
&\quad \mathsf{nat}/\mathsf{base} : (n : \mathbb{N}) \to \overline{n} \approx_{\mathsf{nat}} n
\end{aligned}$$

2) Prelogical relation for commands. Using the logical relation for strictly positive types, we may define the logical relation between syntatic stores and semantic stores:

$$\begin{aligned}
&\textbf{Inductive}\ \sim : \{\Sigma\}\ \mathsf{Store}(\Sigma) \to [\![\Sigma]\!]^{\mathbf{MA}}_{\mathsf{Sig}} \to \mathsf{tp}^+\ \textbf{where}\\
&\quad \mathsf{base} : \mathsf{emp} \sim_{\mathsf{nil}} \star\\
&\quad \mathsf{extend} : \{\Sigma, A, \mathsf{a}, a, \mu, \sigma\} \to (h : \mathsf{val}(\mathsf{a})) \to\\
&\qquad \mathsf{a} \approx_A a \to \mu \sim_\Sigma \sigma \to ((\mathsf{a}, h) :: \mu) \sim_{A::\Sigma} (a, \sigma)
\end{aligned}$$

The prelogical relation for commands is defined relative to a given relation for expressions:

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

Roughly, given a relation R between syntatic and semantic values, a syntactic command m is related to a semantic command m when given logically related syntactic and semantic stores μ and σ, we have that if semantically $m(\sigma)$ computes to a semantic value v and store σ' incurring some cost, then executing (μ, m) will evaluate with the same cost to a syntactic value v such that $R(\mathsf{v}, v)$ and a syntactic store μ' related to the semantic store σ'.

3) Logical relation for expressions. The logical relation for expressions may be defined as in the case of **STLC** (base cases

$$[\![-]\!]^{\mathbf{MA}}_{\mathsf{Exp}} : \{\Sigma, \Gamma, A\}\ \mathsf{Tm}(\Sigma, \Gamma, A) \to (\Sigma' : \mathsf{Sig}) \to \Sigma' \geq \Sigma \to [\![\Gamma]\!]^{\mathbf{MA}}_{\mathsf{Con}}(\Sigma') \to \mathsf{F}([\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma'))$$
$$[\![\mathsf{lam}\{A_1, A_2\}(e)]\!]^{\mathbf{MA}}_{\mathsf{Exp}}(\Sigma', p, \gamma') = \mathsf{ret}(\lambda\Sigma'', p', \lambda a : [\![A_1]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma''). [\![e]\!]^{\mathbf{MA}}_{\mathsf{Exp}}(\Sigma'', \mathsf{tr}(p', p), (a, \mathsf{up}(p', \gamma'))))$$
$$[\![\mathsf{ap}(e, e_1)]\!]^{\mathbf{MA}}_{\mathsf{Exp}}(\Sigma', p, \gamma') = f \leftarrow [\![e]\!]^{\mathbf{MA}}_{\mathsf{Exp}}(\Sigma', p, \gamma'); a \leftarrow [\![e_1]\!]^{\mathbf{MA}}_{\mathsf{Exp}}(\Sigma', p, \gamma); \mathsf{step}^1(f(\Sigma', \mathsf{refl}, a))$$
$$[\![\mathsf{cmd}(m)]\!]^{\mathbf{MA}}_{\mathsf{Exp}}(\Sigma', p, \gamma') = \mathsf{ret}(\lambda\Sigma'', p', (\sigma'' : [\![\Sigma'']\!]^{\mathbf{MA}}_{\mathsf{Sig}}).\ [\![m]\!]^{\mathbf{MA}}_{\mathsf{Cmd}}(\Sigma'', \mathsf{tr}(p', p), \mathsf{up}(p', \gamma'), \sigma''))$$

$$[\![-]\!]^{\mathbf{MA}}_{\mathsf{Cmd}} : \{\Sigma, \Gamma, A\}\ \mathsf{Tm}(\Sigma, \Gamma, A) \to (\Sigma' : \mathsf{Sig}) \to \Sigma' \geq \Sigma \to [\![\Gamma]\!]^{\mathbf{MA}}_{\mathsf{Con}}(\Sigma') \to [\![\Sigma']\!]^{\mathbf{MA}}_{\mathsf{Sig}} \to \mathsf{L}([\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma') \times [\![\Sigma']\!]^{\mathbf{MA}}_{\mathsf{Sig}})$$
$$[\![\mathsf{while}[n](m)]\!]^{\mathbf{MA}}_{\mathsf{Cmd}}(\Sigma', p, \gamma', \sigma') = \mathsf{iter}(g)(\sigma') \textbf{ where}$$
$$g : [\![\Sigma']\!]^{\mathbf{MA}}_{\mathsf{Sig}} \to \mathsf{L}((1 \times [\![\Sigma']\!]^{\mathbf{MA}}_{\mathsf{Sig}}) + [\![\Sigma']\!]^{\mathbf{MA}}_{\mathsf{Sig}})$$
$$g(\sigma) \textbf{ with } \sigma[n] \cdots \mid \mathsf{ff} = \mathsf{step}^1(\mathsf{ret}_{\mathsf{L}}(\mathsf{inl}(\star, \sigma)))$$
$$\cdots \mid \mathsf{tt} = (-, \sigma') \leftarrow_{\mathsf{L}} [\![m]\!]^{\mathbf{MA}}_{\mathsf{Cmd}}(\Sigma', p, \gamma', \sigma); \mathsf{step}^2(\mathsf{ret}(\mathsf{inr}(\sigma')))$$

Fig. 6. The interpretation of expressions and commands of **MA**, selected clauses.

omitted), using the prelogical relation of commands defined in Section V-F2 in the case of reified commands:

$$\approx : \{\Sigma, A\}\ \mathsf{Pg}(\Sigma, A) \to [\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma) \to \mathsf{tp}^+$$
$$\mathsf{e} \approx_{\Sigma, A_1 \Rightarrow A_2} e = \Sigma \mathsf{e}_2 : \mathsf{Tm}(\Sigma, A_1, A_2).\ \mathsf{e} = \mathsf{lam}(\mathsf{e}_2)$$
$$\times \mathsf{U}(\Pi\Sigma'.\ \Pi p : \Sigma' \geq \Sigma.\ \Pi \mathsf{e}_1 : \mathsf{Pg}(\Sigma', A_1), e_1 : [\![A_1]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma').$$
$$\mathsf{e}_1 \approx_{\Sigma', A_1} e_1 \to (\Uparrow^p \mathsf{e})[\mathsf{e}_1] \approx^{\Downarrow}_{\Sigma', A_2} e(\Sigma', p, e_1))$$
$$\mathsf{e} \approx_{\Sigma, \mathsf{cmd}(A)} m = \Sigma \mathsf{m} : \mathsf{Cmd}(\Sigma, A).\ \mathsf{e} = \mathsf{cmd}(\mathsf{m})$$
$$\times \mathsf{U}(\Pi\Sigma'.\ \Pi p : \Sigma' \geq \Sigma.\ (\Uparrow^p \mathsf{m})\ \mathsf{cmd}_{\Sigma', A}(\approx_{\Sigma', A})\ m(\Sigma', p))$$

Following Section III, the operator $-^{\Downarrow}$ lifts a relation on values to computations:

$$-^{\Downarrow} : \{\Sigma, A\}\ (\mathsf{Pg}(\Sigma, A) \to [\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma) \to \mathsf{tp}^+) \to$$
$$(\mathsf{Pg}(\Sigma, A) \to \mathsf{F}([\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma)) \to \mathsf{tp}^+)$$
$$\mathsf{e}\ R^{\Downarrow}\ e = \mathsf{U}(\Pi\ v : [\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma), c : \mathbb{N}.\ e = \mathsf{step}^c(\mathsf{ret}(v)) \to$$
$$(\Sigma \mathsf{v} : \mathsf{Pg}(\Sigma, A).\ (\mathsf{e} \Downarrow^{\eta \bullet^c}_{\P_{\mathsf{E}}} \mathsf{v}) \times \mathsf{v}\ R\ v))$$

We lift the relation to closing instantiations of contexts $\mathsf{Inst}(\Sigma, \Gamma) \coloneqq \mathsf{Sub}_{\Sigma}(\Gamma, \mathsf{nil})$:

$$\textbf{Inductive } \approx : \{\Sigma, \Gamma\}\ \mathsf{Inst}(\Sigma, \Gamma) \to [\![\Gamma]\!]^{\mathbf{MA}}_{\mathsf{Con}}(\Sigma) \to \mathsf{tp}^+ \textbf{ where}$$
$$\mathsf{emp} : \mathsf{emp} \approx_{\Sigma, \mathsf{nil}} \star$$
$$\mathsf{cons} : \{\Sigma, \Gamma, \gamma, \gamma, A, \mathsf{a}, a\}\ \mathsf{a} \approx_{\Sigma, A} a \to \gamma \approx_{\Sigma, \Gamma} \gamma \to$$
$$\mathsf{cons}(\mathsf{a}, \gamma) \approx_{\Sigma, A::\Gamma} (a, \gamma)$$

4) Logical relation for commands. The logical relation for commands is obtained by instantiating the prelogical relation with the logical relation for expressions:

$$\sim : \{\Sigma, A\}\ \mathsf{Cmd}(\Sigma, A) \to$$
$$([\![\Sigma]\!]^{\mathbf{MA}}_{\mathsf{Sig}} \to \mathsf{F}([\![A]\!]^{\mathbf{MA}}_{\mathsf{Ty}}(\Sigma) \times [\![\Sigma]\!]^{\mathbf{MA}}_{\mathsf{Sig}})) \to \mathsf{tp}^+$$
$$m \sim_{\Sigma, A} c = m\ \mathsf{cmd}_{\Sigma, A}(\approx_{\Sigma, A})\ c$$

5) Fundamental theorem of logical relations for adequacy. Using the axioms governing iteration and cost bounds introduced in Section IV, w we may prove the fundamental theorem of the logical relation by mutal induction on the derivation of expressions and commands.

Theorem V.3 (FTLR)**.** *Given an expression $e : \mathsf{Tm}(\Sigma, \Gamma, A)$, if $p : \Sigma' \geq \Sigma$ and $\gamma' \approx_{\Sigma', \Gamma} \gamma'$, then $(\Uparrow^p e)[\gamma'] \approx^{\Downarrow}_{\Sigma', A} [\![e]\!]^{\mathbf{MA}}_{\mathsf{Exp}}(\Sigma', p, \gamma')$. Moreover, given a command $m : \mathsf{Cmd}(\Sigma, \Gamma, A)$, if $p : \Sigma' \geq \Sigma$ and $\gamma' \approx_{\Sigma', \Gamma} \gamma'$, then $(\Uparrow^p m)[\gamma'] \sim_{\Sigma', A} [\![m]\!]^{\mathbf{MA}}_{\mathsf{Cmd}}(\Sigma', p, \gamma')$.*

As a corollary, we obtain cost-aware computational adequacy for both expressions and commands:

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

Extensional adequacy follows immediately:

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

VI. Models of CALF

A model of **calf** in the sense of Niu et al. [2] consists of any locally cartesian closed category $\mathscr{E}$ and an implementation of the associated signature (which we have presented a fragment of in Fig. 1) in $\mathscr{E}$. *op. cit.* constructs an Eilenberg-Moore-type model of **calf** called the *counting model* based on the writer monad associated to a cost monoid $\mathbb{C}$. In the next couple sections we extend the counting model to account for universes and partiality.

VII. A model of CALF*

Fix a presheaf topos $\mathbf{X}$. We will construct a model **calf*** using the internal language of $\mathbf{X}$ — an extensional type theory equipped with (quotient) inductive types and a strict cumulative hierarchy of universes. As part of the input of the model construction, we are given a distinguished proposition

$\mathsf{E} : \Omega$ in $\mathbf{X}$ representing the extensional phase. Following the notation of Niu et al. [2], we write $\bigcirc$ and $\bullet$ for the open and closed modalities associated with E. In the following, let $\alpha < \beta < \gamma$ be universe levels and $\mathbb{C}$ be a cost monoid in the sense of *op. cit.*. Recall that the counting model of **calf** is based on the Eilenberg-Moore adjunction arising from the monad for cost effect $T \coloneqq \bullet\mathbb{C} \times -$, which we adopt for **calf***:

$$\begin{aligned} \mathsf{tp}^+ &: \mathcal{U}_\gamma & \mathsf{tp}^\ominus &: \mathcal{U}_\gamma \\ \mathsf{tp}^+ &= \mathcal{U}_\beta & \mathsf{tp}^\ominus &= \mathsf{Alg}_{\mathcal{U}_\beta}(T) \\ \mathsf{tm}^+(A) &= A & \mathsf{tm}^\ominus(X) &= |X| \end{aligned}$$

Following the notation of *op. cit.*, we write $\mathsf{Alg}_{\mathcal{U}}(T)$ for the type of algebras for the monad T whose carrier is valued in $\mathcal{U}$. Given an algebra $\alpha : \mathsf{Alg}_{\mathcal{U}}(T)$, we write $|\alpha|$ for the carrier and $\alpha.\mathsf{map}$ for the structure map. The value universe can then be modeled as the universe $\mathcal{U}_\alpha : \mathcal{U}_\beta$, and the computation universe is the *trivial algebra* for T valued in $\mathcal{U}_\alpha$-small T-algebras:

$$\begin{aligned} \mathsf{Univ}^+ &: \mathcal{U}_\beta & \mathsf{Univ}^\ominus &: \mathsf{Alg}_{\mathcal{U}_\beta}(T) \\ \mathsf{Univ}^+ &= \mathcal{U}_\alpha & \mathsf{Univ}^\ominus &= \mathsf{trivAlg}(T, \mathsf{Alg}_{\mathcal{U}_\alpha}(T)) \end{aligned}$$

The decoding map for the value universe is the identity: $\mathsf{El}^+(A) = A$. For the computation universe, we can unfold definitions and see that it suffices to define a map $\mathsf{Alg}_{\mathcal{U}_\alpha}(T) \to \mathsf{Alg}_{\mathcal{U}_\beta}(T)$, which is given by the inclusion of $\mathcal{U}_\alpha$-small algebras in $\mathcal{U}_\beta$-small algebras.

Closure under type connectives. One may check that the pair of universes are closed under the connectives of **calf***. Because the internal language of $\mathbf{X}$ is extensional dependent type theory equipped with a strict cumulative universe hierarchy, one can straightforwardly close the pair of universes under the type connectives of **calf*** using the ambient universe.

Inductive types. W-types exist in any topos with a natural numbers object. In particular, this means we may interpret the internal W-type of **calf*** in any presheaf topos.

VIII. A Model of CALF$^\omega$

In this section we extend the construction from Section VII to a model for **calf**$^\omega$.

A. Lifted computations

The type of lifted computations $\mathsf{L}(A)$ are interpreted using the the *quotient inductive-inductive partiality monad* of Altenkirch et al. (written as $(-_\perp, \eta_\perp, \mu_\perp)$):

$$\begin{aligned} \mathsf{L} &: \mathcal{U}_\beta \to \mathsf{Alg}_{\mathcal{U}_\beta}(T) \\ \mathsf{L}(A) &= \alpha_{\mathsf{L}(A)} \end{aligned}$$

The algebra for lifted computations is defined as follows:

$$\begin{aligned} |\alpha_{\mathsf{L}(A)}| &= (TA)_\perp = (\bullet\mathbb{C} \times A)_\perp \\ \alpha_{\mathsf{L}(A)} &: T|\alpha_{\mathsf{L}(A)}| \to |\alpha_{\mathsf{L}(A)}| \\ \alpha_{\mathsf{L}(A)}(c, e) &= (c', a) \leftarrow_\perp e; \eta_\perp(c + c', a) \end{aligned}$$

It is straightforward to verify that the algebra laws are satisfied. The inclusion of free computations is given by $\mathsf{lift} = \eta_\perp$. Sequencing of lifted computations is implemented by threading through the cost of computations:

$$\begin{aligned} \mathsf{bind}_{\mathsf{L}} &: (TA)_\perp \to (A \to (TB)_\perp) \to (TB)_\perp \\ \mathsf{bind}_{\mathsf{L}}(e, f) &= (c_1, a) \leftarrow_\perp e; (c_2, b) \leftarrow_\perp fa; \eta_\perp(c_1 + c_2, b) \end{aligned}$$

IX. Conclusion

In this paper we contribute a family of *cost-aware* metalanguages for studying *intensional* properties via *synthetic* denotational semantics. The metalanguage we present supports synthetic reasoning in two orthogonal directions. First, by basing our work on **calf**, a dependent type theory with an axiomatic theory of the interaction of intension and extension [2], we obtain a rich language for phase-separated constructions. As we show in Sections III-F and V-F, this enables us to formulate and prove cost-aware generalizations of classic Plotkin-type adequacy theorems that restrict immediately to their original extensional counterparts, which improves upon prior work on synthetic denotational semantics in type theory (see Section I-D5). Second, our metalanguage is also synthetic in a more traditional sense by allowing the user to construct conceptually simple denotational semantics of programming languages using only elementary type-theoretic constructions.

We illustrate our approach by proving a cost-aware computational adequacy theorem in the style of Plotkin for the simply-typed lambda calculus and Modernized Algol. These results establish criterions by which cost models for algorithm analysis in **calf** may be deemed to be cost adequate with respect to a given operational semantics, thereby giving a positive answer to the conjecture of Niu et al. [2]. In view of *op. cit.*'s work on algorithm analysis, the metalanguage we have developed constitutes an expressive framework for not only cost-aware programming and verification but also cost-aware metatheory of programming languages.

Future work. In Section I-D4 we mentioned the possibility of extending our work to account for PCF and truly generalizing Plotkin's original adequacy theorem. The main challenge would be to construct an SDT topos with an intrinsic notion of cost structure to obtain a universe of domains that support *arbitrary* fixed-points on endofunctions. In this direction, work on topos-theoretic approaches to program semantics such as [30, 31] will be germane.

Acknowledgement

We are grateful to Jonathan Sterling for productive discussions on the topic of this research, and to Tristan Nguyen at AFOSR for his support.

This work was supported in part by AFOSR under grants MURI FA9550-15-1-0053, FA9550-19-1-0216, and FA9550-21-0009, in part by the National Science Foundation under award number CCF-1901381, and by AFRL through the NDSEG fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR, NSF, or AFRL.

References

[1] G. Plotkin, "Lcf considered as a programming language," *Theoretical Computer Science*, vol. 5, no. 3, pp. 223–255, 1977. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397577900445

[2] Y. Niu, J. Sterling, H. Grodin, and R. Harper, "A cost-aware logical framework," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, Jan. 2022.

[3] R. Harper, *Practical Foundations for Programming Languages*, 1st ed. New York, NY, USA: Cambridge University Press, 2012.

[4] P.-M. Pédrot and N. Tabareau, "The fire triangle: How to mix substitution, dependent elimination, and effects," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, Dec. 2019.

[5] J. Sterling and R. Harper, "Logical relations as types: Proof-relevant parametricity for program modules," *Journal of the ACM*, vol. 68, no. 6, Oct. 2021.

[6] N. Danner, D. R. Licata, and Ramyaa, "Denotational cost semantics for functional languages with inductive types," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, K. Fisher and J. H. Reppy, Eds. Association for Computing Machinery, 2015, pp. 140–151.

[7] G. A. Kavvos, E. Morehouse, D. R. Licata, and N. Danner, "Recurrence extraction for functional programs through call-by-push-value," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, Dec. 2019.

[8] P. W. O'Hearn and R. D. Tennent, Eds., *Algol-like Languages*. Boston, MA: Birkhäuser Boston, 1997, vol. 1.

[9] ——, *Algol-like Languages*. Boston, MA: Birkhäuser Boston, 1997, vol. 2.

[10] J. C. Reynolds, "The Essence of ALGOL," in *Algorithmic Languages: Proceedings of the International Symposium on Algorithmic Languages*, J. W. de Bakker and J. C. van Vliet, Eds. Amsterdam: North-Holland, 1981, pp. 345–372.

[11] F. J. Oles, *Type Algebras, Functor Categories and Block Structure*. USA: Cambridge University Press, 1986, pp. 543–573.

[12] D. R. Ghica, "Slot games: A quantitative model of computation," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 85–97. [Online]. Available: https://doi.org/10.1145/1040305.1040313

[13] D. S. Scott, "Domains for denotational semantics," in *Automata, Languages and Programming*, M. Nielsen and E. M. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 577–610.

[14] J. M. E. Hyland, "First steps in synthetic domain theory," in *Category Theory*, A. Carboni, M. C. Pedicchio, and G. Rosolini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 131–156.

[15] M. P. Fiore and G. Rosolini, "Two models of synthetic domain theory," *Journal of Pure and Applied Algebra*, vol. 116, no. 1, pp. 151–162, 1997.

[16] M. P. Fiore and G. 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*, ser. Lecture Notes in Computer Science, D. van Dalen and M. Bezem, Eds., vol. 1258. Springer, 1996, pp. 129–149.

[17] B. Reus and T. Streicher, "General synthetic domain theory — a logical approach," *Mathematical Structures in Computer Science*, vol. 9, no. 2, pp. 177–223, 1999.

[18] R. E. Møgelberg and M. 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.

[19] M. Paviotti, R. E. Møgelberg, and L. Birkedal, "A model of PCF in Guarded Type Theory," *Electronic Notes in Theoretical Computer Science*, vol. 319, no. Supplement C, pp. 333–349, 2015, the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[20] D. Patterson and A. Ahmed, "The next 700 compiler correctness theorems (functional pearl)," in *International Conference on Functional Programming (ICFP), Berlin, Germany*. Berlin, Germany: ACM Press, 2019.

[21] J. T. Perconti and A. Ahmed, "Verifying an open compiler using multi-language semantics," in *European Symposium on Programming (ESOP)*, Grenoble, France, Apr. 2014.

[22] A. Ahmed, "Verified Compilers for a Multi-Language World," in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, Eds., vol. 32, Asilomar, California, 2015, pp. 15–31.

[23] P. Mates, J. Perconti, and A. Ahmed, "Under control: Compositionally correct closure conversion with mutable state," in *ACM Conference on Principles and Practice of Declarative Programming (PPDP)*, Porto, Portugal, 2019.

[24] N. Benton and C.-K. Hur, "Realizability and compositional compiler correctness for a polymorphic language," Microsoft Research, Tech. Rep. MSR-TR-2010-62, April 2010. [Online]. Available: https://www.microsoft.com/en-us/research/publication/realizability-and-compositional-compiler-correctness-for-a-polymorphic-language/

[25] R. Amadio, N. Ayache, F. Bobot, J. Boender, B. Camp-

bell, I. Garnier, A. Madet, J. McKinna, D. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. Sacerdoti Coen, I. Stark, and P. Tranquilli, "Certified complexity (cerco)," vol. 8552, 08 2014, pp. 1–18.

[26] A. Jung, M. Fiore, E. Moggi, P. W. O'Hearn, J. G. Riecke, G. Rosolini, and I. Stark, "Domains and denotational semantics: History, accomplishments and open problems," *SCHOOL OF COMPUTER SCIENCE RESEARCH REPORTS-UNIVERSITY OF BIRMINGHAM CSR*, 1996.

[27] T. Altenkirch, N. Ghani, P. Hancock, C. McBride, and P. Morris, "Indexed containers," *Journal of Functional Programming*, vol. 25, 2015.

[28] N. Benton, C.-K. Hur, A. J. Kennedy, and C. McBride, "Strongly typed term representations in coq," *Journal of Automated Reasoning*, vol. 49, no. 2, pp. 141–159, 2012. [Online]. Available: https://doi.org/10.1007/s10817-011-9219-0

[29] T. Altenkirch, N. A. Danielsson, and N. Kraus, "Partiality, revisited: The partiality monad as a quotient inductive-inductive type," in *Foundations of Software Science and Computation Structures*, J. Esparza and A. S. Murawski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 534–549.

[30] A. K. Simpson, "Computational Adequacy in an Elementary Topos," in *Computer Science Logic*, G. Gottlob, E. Grandjean, and K. Seyr, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 323–342.

[31] J. Sterling and R. Harper, "Sheaf semantics of termination-insensitive noninterference," in *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. Felty, Ed., vol. 228. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Aug. 2022.