Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 347 (2019) 87–120

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

The Effects of Effects on Constructivism

Liron Cohen[a], Sofia Abreu Faro and Ross Tate[a]

[a] *Department of Computer Science*
Cornell University
Ithaca, NY 14853-7501

Abstract

It is commonly understood that Countable Choice holds constructively due to the underlying computational nature of constructivism. However, in this paper we demonstrate that invoking different notions of computation result in radically different behaviors regarding Countable Choice. In particular, we illustrate that, although *deterministic* computation guarantees Countable Choice, *non*-deterministic computation can negate Countable Choice. We then further show that using *stateful* computation can restore Countable Choice even in the presence of non-determinism. This finding suggests that much of the modern discourse of constructivism assumes a deterministic underlying computational system, despite non-determinism being a fundamental aspect of modern-day computation.

Keywords: constructivism, effects, countable choice, non-determinism, stateful computation

1 Introduction

As software has grown increasingly critical to our society's infrastructure, mechanically verified software has grown increasingly important, feasible, and prevalent. Proof assistants such as Coq [5], Agda [8], Twelf, and Nuprl [1, 14] are some of the most popular tools for mechanical verification. Each of these proof assistants sits on top of a computational system that embodies the mathematical philosophy of constructivism "under which an object exists only if we can construct it" [9]. In proof assistants, construction is done through programs via a tight correspondence to proofs often referred to as the proofs-as-programs paradigm [2].

This paradigm, most famously exhibited by the BHK interpretation, grounds constructivism in computation [43]. Yet there are many different notions of computation. While Turing machines and the λ-calculus crosscut these notions, even they have some subtlety. For example, an algorithm is required to terminate on all inputs, but the definition of termination varies in non-classical settings. One definition gives Markov's Principle [28], where an iterative computation terminates if it cannot proceed indefinitely; another definition gives Bar Induction [12], where a recursive computation terminates if every possible recursion path encounters a

https://doi.org/10.1016/j.entcs.2019.09.006

base case; and there are yet more definitions. Thus it is well known that some constructive principles depend on the specifics of what one considers to entail computation [9].

Nonetheless, some principles are often considered to transcend such details about computation, meaning their verity or falsity is independent of the particular notion of computation employed. A notable example is Countable Choice (CC), which is widely accepted amongst contructivists [7, 9, 25, 32, 33, 36, 44], though not universally [37–41]. Put simply, CC states that any total relation from the natural numbers has a corresponding function exhibiting the totality of that relation. It has been shown that CC holds for any model of type theory standardly constructed from any model of the λ-calculus—more specifically from any partial combinatory algebra [15, 17, 47]—and as such is independent of factors such as a particular definition of termination. In essence, the computation realizing the proof of totality itself describes the desired choice function. This example is particularly important to constructivism because CC is often relied upon to achieve what classically would be done through the Law of Excluded Middle (LEM). For example, without CC and LEM the types (not to be confused with setoids) of (modulated) Cauchy reals are not necessarily Cauchy complete, nor are they necessarily equivalent to the Dedekind reals [27]. CC unifies the most common constructive formulations of the reals, namely the Cauchy, modulated Cauchy, and Dedekind reals [10].

However, in this paper we illustrate that the standard justification for CC makes an implicit assumption about computation, one that underlies much of the discourse of constructivism. That assumption is that the computation used to construct an object (in order to show it exists) is *deterministic*. Indeed, the computational systems underlying every major proof assistant are all deterministic. While some systems such as Coq rely on canonicalization to achieve decidable type-checking, even undecidable extensional systems like Nuprl have coincidentally all made their computational systems deterministic. That is, while there is significant research in constructively *modeling* and reasoning *about* non-deterministic computation, non-deterministic computation has not itself been *directly* incorporated into the computational systems underlying proof assistants.

We show how non-deterministic computation can be soundly incorporated into these computational systems. In fact, the standard computational models of type theory trivially extend to non-deterministic computation—none of the definitions or proofs for these models were truly utilizing the deterministic assumption. However, the same is not true for principles that were derived from these standard models. For example, we show that adding even a modicum of non-determinism, in the form of a possibilistic coin flip, not only makes CC no longer hold in the model, it in fact makes the *negation* of CC hold. This formally supports Schuster's philosophical concern that CC would be incompatible with non-deterministic extensions to constructivism [41].

Non-determinism is just one example of *effectful* computation. Thus, we also consider the impact of *stateful* computation on constructive models. We show that, in addition to still forming a consistent model of type theory, stateful computation

can be used to restore CC *even in the presence of non-determinism*. This is because state can be used to memoize [29] computations. Thus CC can be implemented by memoizing the computation contained in the proof of totality. This suggests that constructive systems wanting to directly support (rather than just model) probabilistic or non-deterministically parallel algorithms should also support some form of state as well in order to be consistent with CC and the unification of the reals.

2 Background

While our discussion will be focused on constructive *type* theory, we want our findings to be relevant to *set* theory as well, and so we must discuss type theories with appropriate expressivenes. Since we will be building multiple such type theories to evaluate the various impacts of effects, to focus on the computational aspects of these theories we rely on known tools to generate the boilerplate. To this end we provide background on topos theory and tripos theory corresponding to models of type/set theory and higher-order logic.

2.1 From Set Theory to Topos Theory

In order to be comparable to common set theories, a type theory needs to exhibit certain important properties. One is that proofs must be irrelevant but not erased, meaning proofs can be used in computations so long as the result of the computation does not depend on the specifics of the proof. This enables functions to correspond to total and determined relations. Another is extensionality of entailment, i.e. that equality on predicates is extensional. This enables the correspondence between functions and total, determined relations to be bijective. Lastly, propositions must be impredicative, meaning there is a type/set (not just universe) of propositions, denoted Ω. This enables the construction of powersets.

Altogether these requirements place us in the setting of (elementary) *topos* theory [23]. Toposes are well established to form models of both set theory and extensional dependent type theories with impredicative propositions [34]. More specifically, since we here focus on *Countable* Choice, we work within the context of W-topos theory, i.e. toposes with a natural-number object modeling the natural numbers.

2.2 From Tripos Theory to Topos Theory

We will be particularly interested in toposes constructed from *triposes* [19, 35]. A tripos is a model of higher-order logic whose type theory is modeled by sets and functions. [1] Higher-order logic has a simple type theory—unit, pairs, and

[1] A tripos is actually a model of higher-order *dependent* predicate logic whose *dependent* type theory is modeled by sets and functions, but the additional dependent structure is irrelevant for our purposes. We use triposes solely in order to construct toposes, and the process for doing so applies to any model of higher-order (simple) logic over any (simple) type theory [20, Corollary 6.1.7].

functions—and a predicate logic formed by $\top$, $\bot$, conjunction, disjunction, implication, equality, and universal and existential quantification. Importantly, the type theory also includes a type, Ω, of propositions, whose terms correspond to propositions in the predicate logic. Thus, higher-order logic provides a means of abstractly reasoning about relations, including impredicative quantification over propositions.

Given a tripos modeling higher-order logic (or more generally a higher-order fibration [20, Definition 5.3.1] [2]), one can construct a topos via the "tripos-to-topos construction" [19]. Whereas the tripos models a simple type theory, the resulting topos is well established to model a dependent type theory. And whereas the tripos might not model extensionality of entailment, the resulting topos necessarily will. Thus the tripos-to-topos construction enables us to work in a simpler setting, with the more complex constructions being automatically generated for us.

Since this construction is standard, we only review the key components. First, an object in the constructed topos is given by a pair $\langle I, \approx_I \rangle$ of a type I and a partial-equivalence relation $\approx_I$ on $I \times I$ in the tripos. Second, a morphism in the constructed topos from $\langle I, \approx_I \rangle$ to $\langle J, \approx_J \rangle$ is a relation R on $I \times J$ that respects $\approx_I$ and $\approx_J$ in the tripos. This relation must be total, meaning $\forall i : I.\ i \approx_I i \supset \exists j : J.\ i\ R\ j$ holds in the tripos, and determined, meaning $\forall i : I, j, j' : J.\ i\ R\ j \wedge i\ R\ j' \supset j \approx_J j'$ holds in the tripos. Furthermore, two morphisms are considered equal if their relations are equivalent in the tripos, effectively baking in extensionality of entailment.

2.3 Realizability Toposes and Triposes

The tripos-to-topos construction is often used for building realizability models [24, 47] of (extensional, impredicative) dependent type theory. In particular, a *realizability topos* is a topos that is constructed from a *realizability tripos*, where a realizability tripos is a tripos that is constructed from a partial combinatory algebra (of codes) through a process we discuss in Section 3.2. The key intuition is that a predicate on a set I specifies for each element i of I which codes (if any) "realize" that the predicate holds for i. This means that an object in the resulting realizability topos is a set I (from the metatheory) along with a relation $i \approx_I i'$ specifying which codes (if any) are considered to realize that i and i' are equal. A common example takes I to be set of natural numbers $\mathbb{N}$ and takes $n \approx_{\mathbb{N}} n'$ to be realized solely by the Church encoding of n when n and n' are equal, and by nothing otherwise. Thus objects in a realizability topos conceptually specify a set I along with a computational interpretation of equality on I.

Another example is the object representing the powerset of natural numbers. For this object the set I is the set of predicates on $\mathbb{N}$ *in the tripos*. The predicate $\approx_I$ states that a code realizes that two predicates ϕ and ψ are equivalent if it can convert any realizer of $\phi(n)$ into the Church encoding of n and it can convert any realizer of $\phi(n)$ into a realizer of $\psi(n)$ and vice versa. Thus two predicates are con-

[2] There is an error in this definition due to a change in terminology across works [21]. The definition should only require a *weak* generic object. This is relevant and evident because realizability triposes have a *strict* generic object, which can only be shown to be weak generic objects. Non-weak generic objects furthermore model extensionality of entailment.

sidered equivalent if they are *computationally* strict, i.e. there is a computation that can extract the natural number for which the realizer holds, and *computationally* equivalent, i.e. there is a computation that can convert between the realizers.

Note that $\approx_I$ is *not* reflexive since there may not be a way to computationally realize that a predicate is strict. Hence, the predicate $i \approx_I i$ is often called the "existence predicate" for i as it indicates that i "exists". The definition of morphisms is designed so that they conceptually need only handle elements that exist according to this existence predicate.

2.4 Relating Topos and Tripos Models of Higher-Order Logic

A topos has an internal model of higher-order logic given by its subobjects, i.e. subsets [20, Corollary 5.4.9]. When a topos is constructed from a tripos, the internal model of the topos is closely related to the associated tripos. In particular, (equivalence classes of) subobjects of $\langle I, \approx_I \rangle$ in the topos bijectively correspond to (equivalence classes of) predicates on I that are strict with respect to $\approx_I$ in the tripos [20, Proposition 6.1.6(ii)]. As such, the interpretations of many propositional connectives, like conjunction, coincide in the two models.

However, there are some differences between these models. For example, the quantification $\forall i : \langle I, \approx_I \rangle.\ \phi(i)$ in the topos corresponds to the quantification $\forall i : I.\ i \approx_I i \supset \phi(i)$ in the tripos, and similarly $\exists i : \langle I, \approx_I \rangle.\ \phi(i)$ in the topos corresponds to $\exists i : I.\ i \approx_I i \wedge \phi(i)$ in the tripos [20, Proposition 6.1.6(iii)]. That is, whenever the topos quantifies over an element $i : \langle I, \approx_I \rangle$, the translation of that quantification in the tripos quantifies over an element of $i : I$ *and insists that* i *"exists"*, i.e. $i \approx_I i$. This step in the translation is particularly important for realizability toposes since it means that proofs of $\forall i : \langle I, \approx_I \rangle.\ \phi(i)$ can have access to a realizer that i "computationally exists", i.e. $i \approx_I i$, and that proofs of $\exists i : \langle I, \approx_I \rangle.\ \phi(i)$ must provide a realizer that i "computationally exists".

2.5 Countable Choice in a Tripos

CC has an internal and an external definition in topos theory that correspond to internal and external CC in set theory [23, 45]. In this paper we discuss *internal* CC because we are concerned about whether it can be used within the theory.

Definition 2.1 (Internal CC for Toposes) *CC holds internally in a W-topos when the following holds in its internal model of higher-order logic for all objects τ:*

$$\forall R : \mathbb{N} \times \tau \to \Omega.\ (\forall n : \mathbb{N}. \exists t : \tau.\ n\ R\ t) \supset \exists f : \mathbb{N} \to \tau. \forall n : \mathbb{N}.\ n\ R\ f(n)$$

In this paper we focus on toposes constructed from triposes, so we focus on this definition's counterpart in tripos theory.

Definition 2.2 (Internal CC for Triposes) *CC holds internally in a tripos when the following holds in its internal model of higher-order logic for all sets I:*

$$\forall R : \mathbb{N} \times I \to \Omega. \quad \mathsf{Tot}(R) \supset \exists S : \mathbb{N} \times I \to \Omega.\ \mathsf{Tot}(S) \wedge S \subseteq R \wedge \mathsf{Det}(S)$$

where

$$\mathsf{Tot}(R) \quad \forall n : \mathbb{N}.\ \mathbb{m}_n \supset \exists i : I.\ n\ R\ i$$

$$\mathbb{m}_n \quad \forall \phi : \mathbb{N} \to \Omega.\ \phi(0) \wedge (\forall n' : \mathbb{N}.\ \phi(n') \supset \phi(n'+1)) \supset \phi(n)$$ [3]

$$S \subseteq R \quad \forall n : \mathbb{N}, i : I.\ n\ S\ i \supset n\ R\ i$$

$$\mathsf{Det}(S) \quad \forall n : \mathbb{N}, i, i' : I.\ n\ S\ i \wedge n\ S\ i' \supset i =_I i'$$

Lemma 2.3 *A W-topos constructed from a tripos internally models CC iff the tripos internally models CC.*

Proof In the case where τ is of the form $\langle I, =_I \rangle$, this follows easily from interpreting Definition 2.1 in the model of strict predicates of the tripos, which is equivalent to the internal model of the topos constructed from that tripos [20, Proposition 6.1.6(ii)]. For τ of the form $\langle I, \approx_I \rangle$, the S given by Definition 2.1 only respects $=_I$, so one then defines $n\ S'\ i$ as $\exists i' : I.\ i' \approx_I i \wedge n\ S\ i'$ to get the appropriate relation that furthermore respects $\approx_I$. □

In the sequel we construct three triposes each based on a different notion of computation with respect to which effects are directly incorporated into the computational model. Using Lemma 2.3, we demonstrate that varying one's notion of computation wildly affects the validity of CC in the resulting constructive type theory.

In order to avoid digressing into low-level details or metatheoretic concerns, we take ZFC [46] as our prevailing metatheory, though we do make a point to note where this particular choice of metatheory is relevant. All of the following lemmas and theorems have been mechanically verified, with more care taken towards metatheoretic concerns, so we refer readers interested in those details to the Coq proofs [13] or appendices.

3 Constructivism and Determinism

Realizability is at the heart of constructivism as it captures the notion of extracting (computable) content from proofs. In turn, partial combinatory algebras [15, 17] are at the heart of realizability as they formalize the key components of computation that serve the proofs-as-programs correspondence. Indeed, a topos is called a realizability topos if it can be derived through a standard construction from a partial combinatory algebra. Due to the properties of this construction and of partial combinatory algebras, every realizability topos models CC [47], supporting the common understanding that CC holds constructively [7,9,25,32,33,36,44]. Next we review partial combinatory algebras and the relevant standard constructions, and we illustrate why CC follows from these foundations of constructivism.

[3] $\mathbb{m}$ is designed so that defining $n \approx_{\mathbb{N}} n'$ as $n =_{\mathbb{N}} n' \wedge \mathbb{m}_n$ makes $\langle \mathbb{N}, \approx_{\mathbb{N}} \rangle$ a natural-number object in the constructed topos.

3.1 Partial Combinatory Algebras

Put simply, a computation accepts inputs and produces outputs. These inputs and outputs can themselves describe computation, i.e. computations are also data. A combinatory algebra formalizes this view, which is critical to developing Turing-complete systems like the λ-calculus, via a set of codes and an ability to apply codes to one another to produce outputs. However, another important aspect of Turing-completeness is that computations may not always manage to actually produce an output, i.e. terminate. A *partial* combinatory algebra incorporates this by permitting application of codes to be partial.

Partial combinatory algebras are formalized in two steps. The first introduces the concepts of codes and application of codes—known as a *partial applicative structure*. The second step then ensures that the partial applicative structure has the necessary expressiveness for modeling computational systems like the λ-calculus.

Definition 3.1 (Partial Applicative Structure) *A partial applicative structure is a set C of "codes" c and a* partial *binary "application" operator $\cdot$ on C. We use $c_f \cdot c_a \downarrow c_r$ to denote c_r being the (successful) result of the application $c_f \cdot c_a$.*

Given a partial applicative structure, one can consider application "expressions" such as $(c_1 \cdot (c_2 \cdot c_3)) \cdot (c_4 \cdot c_5)$. A partial combinatory algebra is a partial applicative structure that is "functionally complete", meaning there is a way to encode such expressions with n free variables as individual codes accepting n arguments through applications.

$$e ::= i \in \mathbb{N} \mid c \in C \mid e \cdot e$$
$$E_n = \{e \mid \text{all } i\text{s in } e \text{ are} < n\}$$

e	$e[c_a]$
0	c_a
$i+1$	i
c	c
$e_f \cdot e_a$	$e_f[c_a] \cdot e_a[c_a]$

$$\frac{}{c \downarrow c} \qquad \frac{e_f \downarrow c_f \qquad e_a \downarrow c_a \qquad c_f \cdot c_a \downarrow c_r}{e_f \cdot e_a \downarrow c_r}$$

Definition 3.2 (Partial Combinatory Algebra) *A partial combinatory algebra (PCA) is a partial applicative structure with an assignment of every expression $e \in E_{n+1}$ to a code $c_{\lambda^n.e} \in C$ that conceptually embodies the λ-calculus term binding the $n+1$ free variables in e, as formalized by the following requirements:*

$$\forall n. \forall e \in E_{n+2}. \forall c_a.\ c_{\lambda^{n+1}.e} \cdot c_a \downarrow c_{\lambda^n.e[c_a]} \qquad \forall e \in E_1. \forall c_a, c_r.\ c_{\lambda^0.e} \cdot c_a \downarrow c_r \iff e[c_a] \downarrow c_r$$

Perhaps the more standard definition of PCAs is as partial applicative structures with S and K *combinators* satisfying certain behaviors [47]. These combinators are simply encodings of particular expressions that are sufficient to ensure that all expressions can be encoded. In our formalization, the S and K combinators are simply the codes $c_{\lambda^2.(0\cdot 2)\cdot(1\cdot 2)}$ and $c_{\lambda^1.0}$ modeling the λ-calculus terms $\lambda x.\lambda y.\lambda z.\ (x\ z)\ (y\ z)$ and $\lambda x.\lambda y.x$, respectively. Similarly, one can define a code c_n^λ that Church-encodes the natural number n:

$$c_0^\lambda = c_{\lambda^1.1} \qquad c_{n+1}^\lambda = c_{\lambda^1.0 \cdot ((c_n^\lambda \cdot 0) \cdot 1)}$$

3.2 Modeling Higher-Order Logic with PCAs

Given a PCA one can construct its corresponding realizability tripos via a standard construction [47]. The core intuition behind a realizability tripos is that a predicate on a set I specifies for each element i which codes from the PCA serve as *realizers* that the predicate holds for i, and that one predicate ϕ entails another ψ when there is a *uniform* code that converts all the realizers of ϕ_i, i.e. $\phi(i)$, to realizers of ψ_i for every i in I. Uniformity means that the code does not itself depend on i—the same code must work for all elements of I.

Uniformity is critical for ensuring entailment corresponds to computation. To see why, consider the fact that in every realizability tripos there is a predicate $\mathbb{m}$ on the natural numbers $\mathbb{N}$ specifying that its only realizer for a natural number n is its Church encoding c^{λ}_{n}. Given a function $f : \mathbb{N} \to \mathbb{N}$ in the metatheory, we can define another predicate on the natural numbers, call it ϕ_f, whose only realizer for a given n is $c^{\lambda}_{f(n)}$. Consider what it means for $\mathbb{m}$ to entail ϕ_f. If entailment could be evidenced by a different code c_n for each $n \in \mathbb{N}$, then $\mathbb{m}$ entails ϕ_f for any function f since c_n could be the constant computation that returns $c^{\lambda}_{f(n)}$. However, requiring a uniform code that works for all indices $n \in \mathbb{N}$ ensures the predicate $\mathbb{m}$ entails ϕ_f if and only if f is computable according to the PCA at hand. Thus uniformity ensures that entailment actually has computational significance.

With these intuitions in mind, we can informally describe how the various propositional connectives are modeled by realizability triposes, with formal descriptions to come as a special case of the more general system in Figure 2. The realizers of a conjunction $\phi_1 \wedge \phi_2$ are simply the Church-encoded pairs of realizers of ϕ_1 and ϕ_2. The realizers of an implication $\phi_1 \supset \phi_2$ are simply the codes that, when applied to a realizer of ϕ_1, necessarily produce a realizer of ϕ_2. There are no realizers for $\bot$, and the realizers of a disjunction $\phi_1 \vee \phi_2$ are the Church-encoded tagged unions of realizers of ϕ_1 and realizers of ϕ_2. A realizer of a universal quantification $\forall i{:}I.\phi_i$ (for inhabited I) is anything that is a realizer of ϕ_i for *every* $i \in I$, whereas a realizer of an existential quantification $\exists i{:}I.\phi_i$ is anything that is a realizer of ϕ_i for *some* $i \in I$. Lastly, any code is a realizer of $\top$, and any code is a realizer of $i =_I i'$ if and only if i and i' are equal in I in the metatheory.

Notice that the realizers for the quantifiers are themselves uniform. That is, a realizer of $\exists i{:}I.\phi_i$ has no *computational* way of knowing *which* i it is a realizer for, and similarly a realizer of $\forall i{:}I.\phi_i$ *cannot* computationally depend on the index i. Thus there is a difference between, for example, realizers of $\forall n{:}\mathbb{N}.\phi_n$ versus realizers of $\forall n{:}\mathbb{N}.\, \mathbb{m}_n \supset \phi_n$. A realizer of the former must be a single code that simultaneously realizes all ϕ_ns, whereas a realizer of the latter is a computation that maps each n to a realizer of ϕ_n.

3.3 Countable Choice in Realizability Triposes

Now we consider Countable Choice with this model of higher-order logic in mind. For this, we introduce a new notation, "$\exists i : I \mid \phi_i.\ \psi_i$", indicating that there internally exists an i in I that *externally* satisfies ϕ_i and *internally* satisfies ψ_i.

A realizer of $\exists i : I \mid \phi_i$. ψ_i is anything that is a realizer of ψ_i for some $i \in I$ satisfying ϕ_i. We also denote $n\ R\ i$ with $R_{\langle n,i\rangle}$, and we use $R_{\langle n,i\rangle}(c)$ to denote that c is a realizer for $R_{\langle n,i\rangle}$.

Lemma 3.3 *CC is equivalent in every realizability tripos to the following holding for every set I:*

$$\forall R : \mathbb{N} \times I \to \Omega.\ \mathsf{Tot}(R) \supset \exists S : \mathbb{N} \times I \to \Omega \mid S \subseteq R \wedge \mathsf{Det}(S).\ \mathsf{Tot}(S)$$

where

$$S \subseteq R = \forall n, i, c.\ S_{\langle n,i\rangle}(c) \implies R_{\langle n,i\rangle}(c) \qquad \mathsf{Det}(S) = \forall n, i, i', c, c'.\ S_{\langle n,i\rangle}(c) \wedge S_{\langle n,i'\rangle}(c') \implies i = i'$$

Proof [13, E-A3] Given Lemma 2.3, this lemma essentially states that inclusion and determinism can be proven computationally if and only if they can be proven in the metatheory. The backwards direction of this is simple. Unfolding definitions, a realizer of inclusions is a code that uniformly converts realizers of $S_{\langle n,i\rangle}$ to realizers of $R_{\langle n,i\rangle}$. If inclusion is provable in the metatheory, then the identity computation exhibits inclusion trivially. Unfolding the definition of determinism, note that the equality predicate $=_I$ is computationally vacuous, meaning the realizers have no computational value beyond whether a realizer exists at all. In this case, $i =_I i'$ has a realizer if and only if i and i' are equal (in I) in the metatheory. Consequently, Det is itself computationally vacuous; it has a realizer if and only if $S_{\langle n,i\rangle}$ and $S_{\langle n,i'\rangle}$ both have realizers for a given n only when i equals i' in the metatheory. Thus, if determinism is provable in the metatheory, then the identity computation exhibits determinism since equality is realized by anything provided the equality holds.

The greater challenge is the forwards direction: showing there is a relation where inclusion and determinism hold in the metatheory whenever there is an appropriate relation where inclusion and determinism are proven computationally. Given a realizer cc of CC and a realizer c^R_{tot} of totality for some relation R, then applying cc to c^R_{tot} necessarily results in a triple of codes c^S_{incl}, c^S_{det}, and c^S_{tot} that realize inclusion, determinism, and totality, respectively, for some relation S. Define a new relation $\hat{S}$ such that $\hat{S}_{\langle n,i\rangle}$ is realized by c when $R_{\langle n,i\rangle}$ is realized by c *and* $S_{\langle n,i\rangle}$ has a realizer (which can be anything). Clearly $\hat{S}$ is included in R in the metatheory. Similarly, $\hat{S}$ is determined in the metatheory because it has realizers for $\langle n, i\rangle$ and $\langle n, i'\rangle$ only when S does, which the existence of c^S_{det} realizing $\mathsf{Det}(S)$ in turn implies that i and i' are equal. Lastly, totality of $\hat{S}$ is realized by the sequential composition of c^S_{tot} realizing totality of S and c^S_{incl} realizing inclusion of S into R. □

Theorem 3.4 *CC is modeled by every realizability tripos.*

Proof [13, E-B1; adapted from 18] By Lemma 3.3, it is sufficient to provide a code that converts realizers that an arbitrary relation R is total into realizers that some metatheoretically-determined subrelation S of R is total. That code is simply the identity computation $c_{\lambda^0.0}$. To see why, note that the definition of S can depend on the specific realizer c^R_{tot} that R is total. By the definition of totality, applying c^R_{tot} to the Church encoding of any natural number n must result in a realizer, say c_n, of $R_{\langle n,i\rangle}$ for some index $i \in I$, *without specifying or even necessarily knowing what i is*. In fact, the returned code might even be a realizer of $R_{\langle n,i\rangle}$ for multiple indices

in I. Let $i_n \in I$ be such a corresponding index for each $n \in \mathbb{N}$.[4] Define $S_{\langle n,i\rangle}$ to be realized by c_n if and only if i equals i_n, trivially making S determined. Since each c_n is a realizer of $R_{\langle n,i_n\rangle}$, S is a subrelation of R. Because application is *deterministic*, applying c^R_{tot} to the Church encoding of n will always result in c_n and hence always be a realizer of $S_{\langle n,i_n\rangle}$, thereby realizing totality. □

Thus all realizability toposes, which by definition are derived from PCAs, necessarily model CC.

4 Introducing Non-Deterministic Computation

Notice that the fact that application is deterministic is critical to the proof that realizability models exhibit CC. However, as we show next, determinism is entirely irrelevant to the realizability interpretation of higher-order logic. Rather, determinism is simply a historical artifact imposing an artificial constraint, and realizability can actually be similarly formulated on the basis of non-deterministic, i.e. *relational*, combinatory algebras. Thus, in this section we develop relational combinatory algebras, illustrate how they naturally still form a model of higher-order logic, and demonstrate that this natural generalization has dramatic effect on constructivism, with something as simple as a coin flip changing CC from being necessarily true to being necessarily false.

4.1 Relational Combinatory Algebras

In order to *directly* model non-deterministic computation, we developed relational combinatory algebras (RCAs), a generalization of PCAs in which the application operator is relational rather than functional. Thus applying one code to another can have zero, one, *or many* possible outcomes.

There is a subtlety to address though. Because PCAs are deterministic, if an application results in some code then that application *always* terminates. However, with RCAs, an application can successfully result in a code on one execution yet fail to terminate on another execution. Thus RCAs need a termination predicate *in addition to* an application relation.

Definition 4.1 (Relational Applicative Structure) *A relational applicative structure is a set C of "codes" c, an "application" relation $c_f \cdot c_a \downarrow c_r$, and a "termination" predicate $c_f \cdot c_a\!\downarrow$ satisfying the following:*

Progress $\forall c_f, c_a.\ c_f \cdot c_a\!\downarrow \implies \exists c_r.\ c_f \cdot c_a \downarrow c_r$

For a relational applicative structure, one can extend the termination predicate $e\!\downarrow$ to applicative expressions:

$$\frac{}{c\!\downarrow} \qquad \frac{e_f\!\downarrow \qquad e_a\!\downarrow \qquad \forall c_f, c_a.\ e_f \downarrow c_f \wedge e_a \downarrow c_a \implies c_f \cdot c_a\!\downarrow}{e_f \cdot e_a\!\downarrow}$$

[4] This assumes CC in the metatheory, which is standard practice here, and which is why we assume specifically ZFC in this paper.

$$c ::= \lambda^n.E_{n+1} \mid \overline{n} \mid p$$
$$p ::= \mathsf{const}_n \mid \mathsf{succ} \mid \mathsf{on}\ \vec{n}\ \mathsf{do}\ p\ \mathsf{else}\ p' \mid \mathsf{flip}$$
$$\vec{n} ::= n, \ldots$$

$$\frac{e[c]\downarrow}{(\lambda^0.e)\cdot c\downarrow} \quad \frac{e[c]\downarrow c'}{(\lambda^0.e)\cdot c\downarrow c'} \quad \frac{}{(\lambda^{n+1}.e)\cdot c\downarrow} \quad \frac{}{(\lambda^{n+1}.e)\cdot c\downarrow \lambda^n.e[c]}$$
$$\frac{}{p\cdot\overline{n}\downarrow} \quad \frac{}{\mathsf{const}_{n'}\cdot\overline{n}\downarrow\overline{n'}} \quad \frac{}{\mathsf{succ}\cdot\overline{n}\downarrow\overline{n+1}} \quad \frac{}{\mathsf{flip}\cdot\overline{n}\downarrow\overline{0}} \quad \frac{}{\mathsf{flip}\cdot\overline{n}\downarrow\overline{1}}$$
$$\frac{n\in\vec{n} \qquad p\cdot\overline{n}\downarrow\overline{n'}}{(\mathsf{on}\ \vec{n}\ \mathsf{do}\ p\ \mathsf{else}\ p')\cdot\overline{n}\downarrow\overline{n'}} \qquad \frac{n\notin\vec{n} \qquad p'\cdot\overline{n}\downarrow\overline{n'}}{(\mathsf{on}\ \vec{n}\ \mathsf{do}\ p\ \mathsf{else}\ p')\cdot\overline{n}\downarrow\overline{n'}}$$

Figure 1. Flip-RCA

Definition 4.2 (Relational Combinatory Algebra) *A relational combinatory algebra is a relational applicative structure with an assignment of expressions $e \in E_{n+1}$ to codes $c_{\lambda^n.e} \in C$ satisfying the followings:*

$$\begin{array}{lll}
\forall n.\forall e \in E_{n+2}.\forall c_a, c_r. & c_{\lambda^{n+1}.e}\cdot c_a \downarrow c_r \implies & c_r = c_{\lambda^n.e[c_a]} \\
\forall e \in E_1.\forall c_a, c_r. & c_{\lambda^0.e}\cdot c_a \downarrow c_r \implies & e[c_a]\downarrow c_r \\
\forall n.\forall e \in E_{n+2}.\forall c_a. & & c_{\lambda^{n+1}.e}\cdot c_a\downarrow \\
\forall e \in E_1.\forall c_a. & e[c_a]\downarrow \implies & c_{\lambda^0.e}\cdot c_a\downarrow
\end{array}$$

Each of these definitions are the straightforward generalizations of PCAs to non-terminating non-deterministic computation. That is, PCAs are simply the special case of RCAs in which application is deterministic and implies termination:

$$\forall c_f, c_a, c_r, c_r'.\ c_f\cdot c_a\downarrow c_r \wedge c_f\cdot c_a\downarrow c_r' \implies c_r = c_r' \qquad \forall c_f, c_a, c_r.\ c_f\cdot c_a\downarrow$$
$$c_r \implies c_f\cdot c_a\downarrow$$

4.2 Modeling Higher-Order Logic with RCAs

Given an RCA one can construct its corresponding *RCA tripos*. All truths in this tripos are still realizable, suggesting that having the term "realizability tripos" refer specifically to PCAs is a misnomer. As such, we introduce the more accurate term "PCA tripos" for that particular notion of realizability.

The core intuition behind an RCA tripos is exactly the same as for a PCA tripos. A predicate on a set I specifies which codes from the RCA serve as realizers that the predicate holds for a particular element i. A predicate ϕ entails another predicate ψ when there is a uniform code that for all $i \in I$ terminates when applied to any realizer of ϕ_i and any possible resulting code is a realizer of ψ_i. The constructions and the proofs are all the same as well, with formal descriptions to come as a special case of the more general system in Figure 2. Thus the deterministic behavior of PCAs is surprisingly irrelevant to their ability to model higher-order logic (and type theory), suggesting that RCAs are actually a more natural fit for realizability theory.

4.3 Refuting Countable Choice with Non-Determinism

This natural generalization of realizability theory, however, has major consequences. In particular, in Figure 1 we present Flip-RCA, an RCA whose corresponding tripos *refutes* CC simply due to the presence of a coin flip.

Flip-RCA is comprised of three key parts. The first is the $\lambda^n.E_{n+1}$ construct,

which describes a λ-value with $n+1$ variables. This construction makes functional completeness trivial, since the code $c_{\lambda^n.e}$ is simply given by $\lambda^n.e$. The termination and reduction rules are standardly defined to guarantee such codes behave as required by functional completeness.

The second key part is the natural-number codes $\overline{n}$ and primitives succ and $\mathsf{on}\ \vec{n}\ \mathsf{do}\ p\ \mathsf{else}\ p'$. These are not strictly necessary, but they make the proof much simpler due to the fact that every "primitive" p only accepts inputs and produces outputs of the form $\overline{n}$. In particular, $\mathsf{on}\ \vec{n}\ \mathsf{do}\ p\ \mathsf{else}\ p'$ emulates the behavior of p on a finite list of inputs $\vec{n}$ and otherwise defers to p'. Also, defining $cast(e)$ as $e \cdot \mathsf{succ} \cdot \overline{0}$ provides an expression that evaluates to $\overline{n}$ whenever the expression e evaluates to the Church encoding of n.

The third key part is flip, the only source of non-determinism in the system that makes this an RCA that is not a PCA. All flip does is non-deterministically evaluate to either $\overline{0}$ or $\overline{1}$.

Lemma 4.3 *Flip-RCA is an RCA.*

Proof [13, E-C2] The $c_{\lambda^n.e}$ codes are given by the $\lambda^n.e$ codes. □

This simple coin flip is enough to refute CC. To demonstrate how, we rely on the following lemma capturing the fact that Flip-RCA describes an extensional finitary computational system. In the lemma we use $\dot{c}$ to informally denote codes with a "primitive hole" such that $\dot{c}[p]$ denotes the code resulting from filling that hole with the primitive p. The formal definitions are tedious and thus omitted here, but can be found in [13, E-C3]. The lemma states that any reduction involving a primitive follows from only finite interactions with that primitive, and so the reduction can proceed similarly for any other primitive that *can* also exhibit those same interactions.

Lemma 4.4 *For all $\dot{c}_f$, $\dot{c}_a$, p, and c_r, such that $\dot{c}_f[p] \cdot \dot{c}_a[p] \downarrow c_r$ holds, there exists a $\dot{c}_r$ satisfying:*

$$c_r = \dot{c}_r[p] \wedge \exists B \subseteq \mathbb{N} \times \mathbb{N}.\ B \text{ is finite } \wedge\ \forall \langle n_i, n_o \rangle \in B.\ p \cdot \overline{n_i} \downarrow \overline{n_o} \\ \wedge \forall p'.\ (\forall \langle n_i, n_o \rangle \in B.\ p' \cdot \overline{n_i} \downarrow \overline{n_o}) \implies \dot{c}_f[p'] \cdot \dot{c}_a[p'] \downarrow \dot{c}_r[p']$$

Proof [13, E-C4] Induction on the proof of $\dot{c}_f[p] \cdot \dot{c}_a[p] \downarrow c_r$. □

Lemma 4.5 *CC is internally equivalent in every RCA tripos to the following holding for every set I:*

$$\forall R : \mathbb{N} \times I \to \Omega.\ \mathsf{Tot}(R) \supset \exists S : \mathbb{N} \times I \to \Omega \mid S \subseteq R \wedge \mathsf{Det}(S).\ \mathsf{Tot}(S)$$

Proof [13, E-A3] Same definitions of $S \subseteq R$ and $\mathsf{Det}(S)$ and proof as with PCAs in Lemma 3.3. □

Theorem 4.6 *There exists a set I for which the negation of CC is internally modeled by the Flip-RCA tripos.*

Proof [13, E-C5] We use $\mathbb{N}$ as the set I for which we prove this negation. RCA (and PCA) triposes model $\neg\phi$ if ϕ has no realizers. Thus it suffices to show that

the existence of a realizer of CC onto $\mathbb{N}$ for Flip-RCA leads to a contradiction. By Lemma 4.5, we can do so by showing there is no code that can convert realizers of totality for relations R on $\mathbb{N}$ into realizers of totality for some metatheoretically-determined subrelation of R.

Suppose cc is such a code. Consider applying cc to $\lambda^0.\mathsf{const}_0 \cdot cast(0)$. The code $\lambda^0.\mathsf{const}_0 \cdot cast(0)$ is a realizer of totality for the relation R^0 whose sole realizer for $R^0_{n,n'}$ is $\overline{0}$ when n' equals 0. Thus this application terminates and results in a realizer c^0_{tot} of totality for some subrelation S^0 of R^0. Since R^0 is a determined relation, one can easily deduce that this implies that c^0_{tot} results in $\overline{0}$ whenever it is applied to a Church encoding of a natural number.

Now define $\dot{c}$ to be $\lambda^0.\bullet \cdot cast(0)$ so that $\dot{c}[\mathsf{const}_0]$ is $\lambda^0.\mathsf{const}_0 \cdot cast(0)$. Lemma 4.4 implies there is a $\dot{c}_{tot}$ and some finite behavior B exhibitable by const_0 such that $\dot{c}_{tot}[\mathsf{const}_0]$ equals c^0_{tot} and applying cc to $\dot{c}[p]$ can reduce to $\dot{c}_{tot}[p]$ whenever p can exhibit behavior B. Let $\vec{n}$ be the list of inputs in B. Then $\mathsf{on}\ \vec{n}\ \mathsf{do}\ \mathsf{const}_0\ \mathsf{else}\ p$ is guaranteed to exhibit behavior B regardless of what p is. Thus applying cc to $\dot{c}[\mathsf{on}\ \vec{n}\ \mathsf{do}\ \mathsf{const}_0\ \mathsf{else}\ p]$ can reduce to $\dot{c}_{tot}[\mathsf{on}\ \vec{n}\ \mathsf{do}\ \mathsf{const}_0\ \mathsf{else}\ p]$ for any primitive p.

There are two particularly important primitives to consider. One is primitives of the form const_m, in which case $\dot{c}[\mathsf{on}\ \vec{n}\ \mathsf{do}\ \mathsf{const}_0\ \mathsf{else}\ \mathsf{const}_m]$ is a realizer of totality for the relation $R^{\vec{n};m}$ whose sole realizer for $R^{\vec{n};m}_{n,n'}$ is $\overline{0}$ when n is in $\vec{n}$ and n' equals 0, or $\overline{m}$ when n is not in $\vec{n}$ and n' equals m. This implies that $\dot{c}_{tot}[\mathsf{on}\ \vec{n}\ \mathsf{do}\ \mathsf{const}_0\ \mathsf{else}\ \mathsf{const}_m]$ is a realizer of totality for some subrelation $S^{\vec{n};m}$ of $R^{\vec{n};m}$. Again, since $R^{\vec{n};m}$ is a determined relation, one can easily deduce that this implies that $c^{\vec{n};m}_{tot} = \dot{c}_{tot}[\mathsf{on}\ \vec{n}\ \mathsf{do}\ \mathsf{const}_0\ \mathsf{else}\ \mathsf{const}_m]$ results in $\overline{m}$ whenever it is applied to a Church encoding of a natural number not in $\vec{n}$.

The other important case is the primitive flip. Then, $\dot{c}[\mathsf{on}\ \vec{n}\ \mathsf{do}\ \mathsf{const}_0\ \mathsf{else}\ \mathsf{flip}]$ is a *non-deterministic* realizer of totality for the relation $R^{\mathbb{N}}$ whose sole realizer for $R^{\mathbb{N}}_{n,n'}$ is $\overline{n'}$ for *any* n. This implies that $c^{\mathsf{flip}}_{tot} = \dot{c}[\mathsf{on}\ \vec{n}\ \mathsf{do}\ \mathsf{const}_0\ \mathsf{else}\ \mathsf{flip}]$ is a realizer of totality for some determined subrelation $S^{\mathbb{N}}$ of $R^{\mathbb{N}}$. Now consider what happens when we apply c^{flip}_{tot} to some n *not* in $\vec{n}$. We know that $c^{\vec{n};m}_{tot} \cdot \overline{n}$ evaluates to $\overline{m}$. Since flip *can* recreate the input-output behaviors of both const_0 and const_1, Lemma 4.4 implies that $c^{\mathsf{flip}}_{tot} \cdot \vec{n}$ *can* evaluate to *both* $\overline{0}$ and $\overline{1}$. This means that $\overline{0}$ must realize $S^{\mathbb{N}}_{n,n_0}$ for some n_0, and similarly $\overline{1}$ must realize $S^{\mathbb{N}}_{n,n_1}$ for some n_1. Since $\overline{0}$ can only realize $R^{\mathbb{N}}_{n,n_0}$ when n_0 equals 0, and $S^{\mathbb{N}}$ is a subrelation of $R^{\mathbb{N}}$, this implies n_0 must equal 0, and similarly n_1 must equal 1. Thus both $S^{\mathbb{N}}_{n,0}$ and $S^{\mathbb{N}}_{n,1}$ are realizable. Since the assumed behavior of cc implies that $S^{\mathbb{N}}$ is determined, this implies 0 equals 1, thereby producing a contradiction. □

Interestingly, this proof can easily be modified to show that Flip-RCA refutes even *Weak* Countable Choice, which states that choice is possible if there is at most one choice to be made across all the countable inputs [10]. Weak Countable Choice is sufficient to unify the various formulations of the reals [10], thus suggesting that the Cauchy, modulated Cauchy, and Dedekind reals might indeed be distinct in the topos for Flip-RCA.

5 Introducing Stateful Computation

We have shown that, although non-determinism naturally fits into realizability models of higher-order logic, a flip of a coin can invalidate CC despite it holding so trivially before. Now we demonstrate that further extending the computation system with mutable state can restore CC even in the presence of non-determinism. This means that CC is not wholly incompatible with non-determinism, contrary to Schuster's concern [41].

5.1 Stateful Combinatory Algebras

In order to *directly* model stateful computation, we developed stateful combinatory algebras (SCAs), a generalization of RCAs in which the application operator is stateful. That is, applying one code to another requires a state that it can then mutate. Just as there are PCAs that can *model* non-deterministic computation [42], PCAs can *model* state by using, say, the state monad [31]. But it is impossible to *force* PCA computations to share state—the requirements for the S combinator force it to duplicate any state a PCA computation might be using. SCAs ensure that all computation operates on the same mutating state, which, as we show, is a critical component in their ability to implement CC even in the presence of non-deterministic computation.

Definition 5.1 (Stateful Applicative Structure) *A stateful applicative structure is an inhabited set Σ of "states" σ, a "possible future" preorder $\sigma \leq \sigma'$, a set C of "codes" c,*[5] *an "application" relation $c_f \cdot c_a \downarrow^{\sigma}_{\sigma'} c_r$, and a "termination" predicate $c_f \cdot c_a \downarrow^{\sigma}$ satisfying the following properties:*

Preservation $\forall \sigma, c_f, c_a, \sigma', c_r.\ c_f \cdot c_a \downarrow^{\sigma}_{\sigma'} c_r \implies \sigma \leq \sigma'$

Progress $\forall \sigma, c_f, c_a.\ c_f \cdot c_a \downarrow^{\sigma} \implies \exists \sigma', c_r.\ c_f \cdot c_a \downarrow^{\sigma}_{\sigma'} c_r$

The concept of "possible futures" here captures the fact that, even in a system with mutable state, the system can maintain certain invariants about its state and how it progresses, as enforced by the preservation property. These invariants will be critical to implementing CC. Note, though, that the application relation and termination predicate are not themselves necessarily preserved by futures; an application is permitted to reduce to a code in a given state that it cannot reduce to in a future state, and a termination only guarantees that the *current* state *can* be mutated to provide a result. Thus this is not simply a standard possible-worlds structure [26].

We extend the definitions of application $e \downarrow^{\sigma}_{\sigma'} c_r$ and termination $e \downarrow^{\sigma}$ to applicative expressions as follows:

[5] Our Coq formalization [13, C-D1] also permits one to specify a "validity" predicate $\sigma \vdash c$ indicating which codes are valid in which states. Here we elide this additional degree of control as it is irrelevant for the current discussion.

$$\frac{}{c \downarrow^{\sigma}_{\sigma} c} \qquad \frac{e_f \downarrow^{\sigma}_{\sigma'} c_f \qquad e_a \downarrow^{\sigma'}_{\sigma''} c_a \qquad c_f \cdot c_a \downarrow^{\sigma''}_{\sigma'''} c_r}{e_f \cdot e_a \downarrow^{\sigma}_{\sigma'''} c_r}$$

$$\frac{}{c\downarrow^{\sigma}} \qquad \frac{e_f\downarrow^{\sigma} \qquad \forall \sigma', c_f.\ e_f \downarrow^{\sigma}_{\sigma'} c_f \implies e_a\downarrow^{\sigma'} \wedge \forall \sigma'', c_a.\ e_a \downarrow^{\sigma'}_{\sigma''} c_a \implies c_f \cdot c_a\downarrow^{\sigma''}}{e_f \cdot e_a\downarrow^{\sigma}}$$

Definition 5.2 (Stateful Combinatory Algebra) *A stateful combinatory algebra is a stateful applicative structure with an assignment of every expression $e \in E_{n+1}$ to a code $c_{\lambda^n.e} \in C$ satisfying the following properties in all states $\sigma, \sigma' \in \Sigma$:*

$$\begin{array}{lll}
\forall n. \forall e \in E_{n+2}. \forall c_a, c_r. & c_{\lambda^{n+1}.e} \cdot c_a \downarrow^{\sigma}_{\sigma'} c_r \implies & \sigma' = \sigma \wedge c_r = c_{\lambda^n.e[c_a]} \\
\forall e \in E_1. \forall c_a, c_r. & c_{\lambda^0.e} \cdot c_a \downarrow^{\sigma}_{\sigma'} c_r \implies & e[c_a] \downarrow^{\sigma}_{\sigma'} c_r \\
\forall n. \forall e \in E_{n+2}. \forall c_a. & & c_{\lambda^{n+1}.e} \cdot c_a\downarrow^{\sigma} \\
\forall e \in E_1. \forall c_a. & e[c_a]\downarrow^{\sigma} \implies & c_{\lambda^0.e} \cdot c_a\downarrow^{\sigma}
\end{array}$$

RCAs are the special case of SCAs with precisely one state.

5.2 Modeling Higher-Order Logic with SCAs

Since PCAs and RCAs are each special cases of SCAs, we were informal about how they model impredicative higher-order logic. Now we provide a formal description of the model in Figure 2. One technical note is that types in our model are *inhabited* sets. This technically means that our model specifies a higher-order fibration [20, Definition 5.3.1] [6], which is a generalization of a tripos. We do this because it permits a simpler interpretation of universal quantification. Furthermore, the standard tripos-to-topos construction works for any higher-order fibration [20, Corollary 6.1.7], so the applicability to set theory and type theory is maintained. In fact, the resulting topos is equivalent to the topos that would be derived from the tripos construction.

Theorem 5.3 *For any SCA, Figure 2 specifies a consistent model of higher-order logic.*

Proof [13, D-C1] The remaining components and proofs for a higher-order fibration follow easily from the definitions in Figure 2. The only thing we prove explicitly here is consistency.

A proposition ϕ in Figure 2 is realizable if there exists a state σ and code c_ϕ such that $\phi^\sigma(c_\phi)$ holds. The progress property of SCAs and the definition of entailment in Figure 2 imply that when a proposition entails another one, realizability of the former implies realizability of the latter. Since the proposition $\top$ is realizable and the proposition $\bot$ is not, $\top$ cannot entail $\bot$, guaranteeing consistency of the model for any SCA. □

[6] There is an error in this definition due to a change in terminology across works [21]. The definition should only require a *weak* generic object. This is relevant and evident because realizability triposes have a *strict* generic object, which can only be shown to be weak generic objects. Non-weak generic objects furthermore model extensionality of entailment.

Notation

$c_f \cdot c_a \downarrow^{\sigma} \phi \triangleq c_f \cdot c_a \downarrow^{\sigma} \wedge \forall \sigma', c_r.\ \ c_f \cdot c_a \downarrow^{\sigma}_{\sigma'} c_r \implies \phi^{\sigma'}(c_r)$

Fibration

Type A type τ is an inhabited set.

Context A context Γ is an inhabited set.

Proposition A proposition ϕ is a "stateful" predicate on codes $\phi^{\sigma}(c)$ that is "future-stable":

$$\forall \sigma, \sigma', c.\ \ \sigma \leq \sigma' \wedge \phi^{\sigma}(c) \implies \phi^{\sigma'}(c)$$

Predicate A predicate ϕ in context Γ assigns to each inhabitant γ of Γ a proposition ϕ_γ.

Entailment A predicate ϕ entails a predicate ψ in Γ if there exists a code c satisfying the following:

$$\forall \gamma \in \Gamma. \forall \sigma, c_\phi.\ \ \phi^{\sigma}_{\gamma}(c_\phi) \implies c \cdot c_\phi \downarrow^{\sigma}_{\psi_\gamma}$$

Substitution A substitution t from a context Γ to a context Γ' is a function from Γ to Γ'. For a predicate ϕ in context Γ', the substituted predicate $\phi[t]$ in Γ is realized by c for γ in σ when $\phi^{\sigma}_{t(\gamma)}(c)$ holds.

Types

Unit The type $\mathbf{1}$ is the singleton set $\mathbf{1}$.

Product The type $\tau \times \tau'$ is the set of pairs $\tau \times \tau'$.

Function The type $\tau \to \tau'$ is the set of functions $\tau \to \tau'$.

Impredicativity The type Ω is the set of future-stable stateful predicates on codes $\{\phi \subseteq \Sigma \times C \mid \phi \text{ is future-stable}\}$.

Predicates

Top The predicate $\top$ in Γ is realized by (any) c for γ in σ.

Conjunction The predicate $\phi \times \psi$ in Γ is realized by c for γ in σ if the following holds:

$$\forall \sigma'.\ \ \sigma \leq \sigma' \implies c \cdot c_{\lambda^1.1} \downarrow^{\sigma'}_{\psi_\gamma} \phi_\gamma \wedge c \cdot c_{\lambda^1.2} \downarrow^{\sigma'}$$

Bottom The predicate $\bot$ in Γ has no realizers for γ in any σ.

Disjunction The predicate $\phi \vee \psi$ in Γ is realized by $c_{\lambda^1.1 \cdot c_\phi}$ for γ in σ if $\phi^{\sigma}_{\gamma}(c_\phi)$ holds, and by $c_{\lambda^1.2 \cdot c_\psi}$ for γ in σ if $\psi^{\sigma}_{\gamma}(c_\psi)$ holds.

Implication The predicate $\phi \supset \psi$ in Γ is realized by c for γ in σ if the following holds:

$$\forall \sigma', c_\phi.\ \ \sigma \leq \sigma' \wedge \phi^{\sigma'}_{\gamma}(c_\phi) \implies c \cdot c_\phi \downarrow^{\sigma'}_{\psi_\gamma}$$

Equality The predicate $=_\tau$ in context $\Gamma \times (\tau \times \tau)$ is realized by (any) c for $\langle \gamma, \langle x, y \rangle \rangle \in \Gamma \times (\tau \times \tau)$ in σ if x equals y.

Universal Quantification For a predicate ϕ in a context $\Gamma \times \tau$, the predicate $\forall \tau.\phi$ in Γ is realized by c for γ in σ if $\forall x \in \tau.\ \phi^{\sigma}_{\langle \gamma, x \rangle}(c)$ holds.

Existential Quantification For a predicate ϕ in a context $\Gamma \times \tau$, the predicate $\exists \tau.\phi$ in Γ is realized by c for γ in σ if $\exists x \in \tau.\ \phi^{\sigma}_{\langle \gamma, x \rangle}(c)$ holds.

Figure 2. SCA Model of Higher-Order Logic

5.3 Restoring Countable Choice with State

Next we show that introducing state enables SCA triposes to model CC even in the presence of non-determinism. In particular, we use state to memoize realizers of totality. Memoization [29] (whose original intent was to optimize computation) is the method of wrapping a computation with something that keeps track of inputs already passed to this computation and their corresponding outputs. Most importantly for our purposes, it has the benefit of always providing the same output for a given input even when the generating computation is itself non-deterministic. While we could provide a general proof that a tripos for any SCA with a special memoizing combinator models CC, due to space constraints we simply provide a concrete example of such an SCA. Thus Figure 3 defines Mem-SCA, whose ndnat code provides non-determinism, and whose memo code implements memoization (via lookup codes).

Mem-SCA is presented in two parts. We describe the left-hand side of Figure 3 first, which formalizes *pre*-states and a "frozen" computational system under a given pre-state ς. A pre-state is comprised of an "allocation" table α and a "memoization" table μ. These tables do not necessarily satisfy the invariants of the system that will enable Mem-SCA to model CC, but they are sufficient for specifying a computational system. An entry $\langle \ell, c \rangle$ in an allocation table α indicates that the memoizations at location ℓ should be generated by code c. An entry $\langle \ell, n, c \rangle$ in a memoization table μ

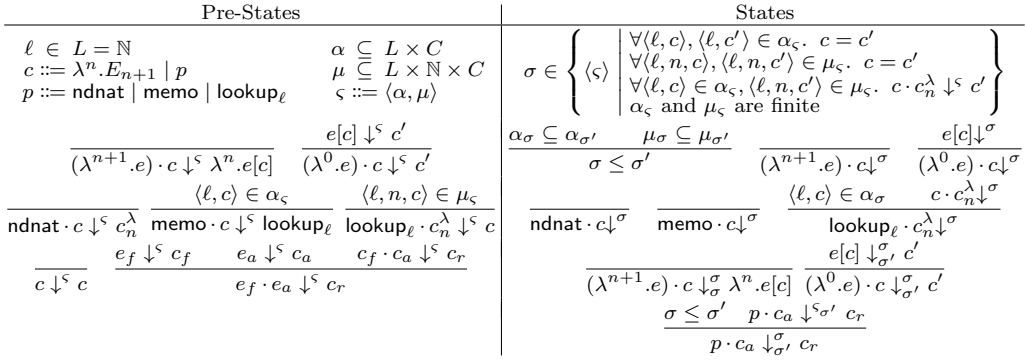

Figure 3. Mem-SCA

indicates that the input n has a memoized output c at location ℓ. Consequently, the computational system specifies that memo applied to a code c can reduce to a lookup_ℓ for any location ℓ whose entries should be generated by c according to α. Similarly, the computational system specifies that lookup_ℓ applied to a Church-encoded natural number n can reduce to a memoized ouput c for input n at location ℓ according to μ. The remainder of the computational system behaves as expected for the λ-calculus, with the addition that ndnat can reduce to any Church-encoded natural number.

Note that a pre-state does not actually guarantee that the entries in its memoization table are generated according to its allocation table, nor does it have any notion of mutating state. Both of these issues are addressed by the right-hand side of Figure 3, which formalizes states and a mutating computational system. A state is a pre-state satisfying additional invariants. First and second, a given location can be generated by at most one code and can have at most one memoized output for each given input. Third, every memoized output is indeed a possible result of applying the input to the generating code. Last, the tables are finite. A state's possible futures are simply all states containing its entries (and possibly more). Termination is straightforward, with the one subtlety that lookup_ℓ only terminates on inputs that its generating code terminates on. Lastly, mutating reduction does not actually prescribe how the state should be mutated—instead, its effect is that the state can be mutated to any future state *provided* the application would reduce according to the frozen computational system, i.e. provided the future state has enough entries that the application can reduce without needing to add more entries. This means that the issue of determining actually how to mutate the state such that the reduction can be completed is delegated to the proof of progress.

Lemma 5.4 *Mem-SCA is an SCA.*

Proof [13, E-D2] We provide the only interesting aspect of this proof, which is progress. For this, we rely on the fact that $c_f \cdot c_a \downarrow^\varsigma c_r$ is easily shown to imply $c_f \cdot c_a \downarrow^{\varsigma'} c_r$ whenever ς' contains all the entries in ς. Progress is proven by induction on the proof of termination, the only interesting cases for which are memo and lookup_ℓ.

Suppose $\mathsf{memo} \cdot c_a{\downarrow}^{\sigma}$ holds. We need to provide a state σ' and a code c_r such that $\mathsf{memo} \cdot c_a \downarrow^{\sigma}_{\sigma'} c_r$ holds. Since α_σ is finite, there is an "unused" location ℓ. Define ς' to be ς_σ with $\langle \ell, c \rangle$ added to the allocation table, and define c_r to be lookup_ℓ. The required reduction and the fact that the pre-state ς' satisfies the requirements to provide a state σ' follow easily.

Now suppose $\mathsf{lookup}_\ell \cdot c_a{\downarrow}^{\sigma}$ holds. We need to provide a state σ' and a code c_r such that $\mathsf{lookup}_\ell \cdot c_a \downarrow^{\sigma}_{\sigma'} c_r$ holds. The assumption implies that c_a is the Church encoding of some natural number n, that ℓ has a corresponding code c_f in the allocation table α_σ, and that $c_f \cdot c_n^{\lambda}{\downarrow}^{\sigma}$ holds. Since μ_σ is finite we can check to see if it has a code c corresponding to ℓ and n, in which case σ' is simply σ and c_r is simply c. Otherwise, by the induction hypothesis $c_f \cdot c_n^{\lambda}{\downarrow}^{\sigma}$ entails the existence of a state $\hat{\sigma}$ and code $\hat{c}$ such that $c_f \cdot c_n^{\lambda} \downarrow^{\sigma}_{\hat{\sigma}} \hat{c}$ holds. Again, since $\mu_{\hat{\sigma}}$ is finite we can check to see if it has a code c' corresponding to ℓ and n, in which case σ' is simply $\hat{\sigma}$ and c_r is simply c'. Otherwise, we define ς' to be $\varsigma_{\hat{\sigma}}$ with $\langle \ell, \hat{c} \rangle$ added to the allocation table, and we define c_r to be $\hat{c}$. The required reduction and the fact that the pre-state ς' satisfies the requirements to provide a state σ' follow easily. □

Lemma 5.5 *CC is internally equivalent in every SCA tripos to the following holding for every set I:*

$$\forall R : \mathbb{N} \times I \to \Omega.\ \mathsf{Tot}(R) \supset \exists S : \mathbb{N} \times I \to \Omega \mid S \subseteq R \wedge \mathsf{Det}(S).\ \mathsf{Tot}(S)$$

where

$$S \subseteq R : \quad \forall n, i, \sigma, c.\ S^{\sigma}_{\langle n,i \rangle}(c) \implies R^{\sigma}_{\langle n,i \rangle}(c)$$

$$\mathsf{Det}(S) : \quad \forall n, i, i', \sigma, c, c'.\ S^{\sigma}_{\langle n,i \rangle}(c) \wedge S^{\sigma}_{\langle n,i' \rangle}(c') \implies i = i'$$

Proof [13, E-A5] The reasoning is very similar to PCAs and RCAs except for one nuance with state. One applies *cc* to c^R_{tot} as before, now in some given state σ, to again get a triple of codes c^S_{incl}, c^S_{det}, and c^S_{tot}, and a future state σ'. As before, define the new relation $\hat{S}$ such that $\hat{S}^{\sigma''}_{\langle n,i \rangle}$ is realized by c when $R^{\sigma''}_{\langle n,i \rangle}$ is realized by c, $S^{\sigma''}_{\langle n,i \rangle}$ has a realizer, *and* σ'' is a possible future of σ'. This final requirement addresses the fact that c^S_{incl}, c^S_{det}, and c^S_{tot} are only guaranteed to exhibit their expected behaviors in state σ' *and any of its possible futures* due to the future-stability of propositions. Consequently, the remainder of the proof can proceed as before. □

Note that the definition of $\mathsf{Det}(S)$ in Lemma 5.5 requires realizability of $S_{\langle n,i \rangle}$ and $S_{\langle n,i' \rangle}$ to imply that i and i' are equal *only if* they are realizable *in the same state* σ. Thus it is perfectly acceptable for $S_{\langle n,i \rangle}$ and $S_{\langle n,i' \rangle}$ to be realizable for distinct i and i' in distinct states. Furthermore, totality of S only requires $S_{\langle n,i \rangle}$ to *eventually* be a realizable for some i for each n. Thus S can be non-deterministic *across states* and be only finitely defined at any particular state, which is how we address the challenges of non-determinism.

Theorem 5.6 *CC is internally modeled by the tripos for Mem-SCA.*

Proof [13, E-D3] The realizer of the proposition in Lemma 5.5 for Mem-SCA is simply memo. Suppose c^R_{tot} is a realizer of totality in a given state σ for a relation R. We need to show that $\mathsf{memo} \cdot c^R_{tot}$ terminates in σ, and that any code it can reduce

to in a possible future σ' is necessarily a realizer of totality in σ' of some determined subrelation of R. Termination is trivial since memo is defined to terminate on all inputs in all states, so reduction is the primary challenge.

By definition, $\mathsf{memo} \cdot c^R_{tot}$ only reduces to lookup_ℓ and only in states σ' for which $\langle \ell, c^R_{tot} \rangle$ is in $\alpha_{\sigma'}$. Thus we need to show that lookup_ℓ is a realizer of totality in σ' for some determined subrelation of R. Let $c_n^{\sigma''}$ denote codes for which $\langle \ell, n, c_n^{\sigma''} \rangle \in \mu_{\sigma''}$ and $\sigma' \leq \sigma''$ hold, which (if it exists) is necessarily unique for a given σ''. Note that, if $c_n^{\sigma''}$ exists, then it is a realizer of $R^{\sigma''}_{\langle n,i \rangle}$ for some i. The reason is that, by the required properties of states, in order to be in the memoization table for ℓ the code $c_n^{\sigma''}$ must be a possible result of applying the generator for ℓ (as specified by α) to c_n^λ. By assumption, that generator is c^R_{tot} and consequently a realizer of totality of R, so its output on input c_n^λ is necessarily a realizer of $R^{\sigma''}_{\langle n,i \rangle}$ for some i. So let $i_n^{\sigma''}$ be a selection of indices in I such that $c_n^{\sigma''}$ is a realizer of $R^{\sigma''}_{\langle n, i_n^{\sigma''} \rangle}$, and such that $i_n^{\sigma'''}$ equals $i_n^{\sigma''}$ whenever both are defined and σ''' is a possible future of σ'' (and so $c_n^{\sigma'''}$ equals $c_n^{\sigma''}$). [7]

Given these choices, define $S^{\sigma''}_{\langle n,i \rangle}(c)$ to hold when c equals $c_n^{\sigma''}$ and when i equals $i_n^{\sigma''}$ (and both $c_n^{\sigma''}$ and $i_n^{\sigma''}$ exist). Each proposition $S_{\langle n,i \rangle}$ is future-stable because $c_n^{\sigma''}$ is future-stable and each $i_n^{\sigma''}$ was chosen to be future-stable. Since we already established that $c_n^{\sigma''}$ is a realizer of $R^{\sigma''}_{\langle n, i_n^{\sigma''} \rangle}$, S is a subrelation of R. By definition, $S^{\sigma''}_{\langle n,i \rangle}$ and $S^{\sigma''}_{\langle n,i' \rangle}$ are both realizable in a given state σ'' only when both equal $i_n^{\sigma''}$, ensuring determinism of S. It remains to prove that lookup_ℓ is a realizer of totality for S. Since c^R_{tot} is a realizer of totality in σ', it terminates on all Church-encoded natural-number inputs in σ', which implies lookup_ℓ does as well. Any code that can result from applying lookup_ℓ to a c_n^λ in a possible future σ'' is necessarily in the memoization table for ℓ in σ'' and therefore equal to $c_n^{\sigma''}$, which by definition is a realizer of $S_{\langle n, i_n^{\sigma''} \rangle}$ in σ''. Thus lookup_ℓ is a realizer of totality for a determined subrelation of R, and hence memo is a realizer of the proposition in Lemma 5.5, thereby evidencing that the tripos for Mem-SCA models CC. □

Although we do not formally develop it here, a similar SCA can even model a principle known as dependent choice (DC), which is strictly stronger than CC [4,22]. The state provides a table in which each entry of the required sequence is simply generated on demand from the previous one according to the allocated realizer of totality. As with Mem-SCA, this works even in the presence of non-determinism.

6 Related Work and Conclusions

This paper demonstrates that key principles of constructivism highly depend on the effectful notion of the computation system, using CC as an illustrative example. We show that the traditional constructive proof of CC fundamentally relies upon a deterministic computational system, and that adding even a coin flip entails its

[7] This assumes Zorn's Lemma in the metatheory, which is why we assume specifically ZFC in this paper. The Coq proof provides a more careful construction that, as is standard practice here, assumes only countable choice in the metatheory [13, E-D3].

negation. We further show that then adding mutable state to the computational system makes it again possible to implement CC. In doing so, the paper extends the boundaries of constructivism towards a truly proofs-as-programs paradigm, not just a proofs-as-*deterministic*-programs paradigm.

This paper focuses on the effect of effects on *existing* principles of constructivism. Other works similarly each make some choice principle compatible with computations with continuations [3, 16, 30]. Interestingly, despite the difference in goals, these systems use techniques as coinduction, lazy evaluation, and infinite terms that are employed in a manner bearing resemblance to our memoization technique.

It would also be interesting to explore what *new* principles might be made possible by effects. For example, Bickford et al. [6] explore using the stateful nature of Nuprl's library system to provide free-choice sequences [11]. We suspect these techniques can be combined to model both CC and free choice, and even extended to support notions of choice almost reaching ZFC.

References

[1] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in Computational Type Theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[2] Joseph L. Bates and Robert L. Constable. Proofs as Programs. *Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[3] Stefano Berardi, Marc Bezem, and Thierry Coquand. On the Computational Content of the Axiom of Choice. *Journal of Symbolic Logic*, 63(2):600–622, 1998.

[4] Paul Bernays. A System of Axiomatic Set Theory. Part III. Infinity and Enumerability. Analysis. *Journal of Symbolic Logic*, 7(2):65–89, 1942.

[5] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development.* Springer-Verlag Berlin Heidelberg, 2004.

[6] Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. Computability beyond Church-Turing via Choice Sequences. In *Logic in Computer Science*, pages 245–254, 2018.

[7] Errett Bishop and Douglas Bridges. *Constructive Analysis.* Springer-Verlag Berlin Heidelberg, 1985.

[8] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*, pages 73–78, 2009.

[9] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics.* London Mathematical Society Lecture Notes Series. Cambridge University Press, 1987.

[10] Douglas Bridges, Fred Richman, and Peter Schuster. A Weak Countable Choice Principle. *Proceedings of the American Mathematical Society*, 128(9):2749–2752, 2000.

[11] L. E. J. Brouwer. Begründung der Mengenlehre unabhängig vom logischen Satz vom ausgeschlossenen Dritten. Zweiter Teil: Theorie der Punktmengen. *Verhandelingen der Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam (Eerste Sectie)*, 12(7), 1919.

[12] L. E. J. Brouwer. *Brouwer's Cambridge Lectures on Intuitionism.* Cambridge University Press, 1981.

[13] Liron Cohen, Sofia Abreu Faro, and Ross Tate. The Effects of Effects on Constructivism: Coq Proof. preprint, 2019. URL: `https://www.cs.cornell.edu/~ross/publications/effectful/`

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

[15] Solomon Feferman. A Language and Axioms for Explicit Mathematics. In *Algebra and Logic*, pages 87–139, 1975.

[16] Hugo Herbelin. A Constructive Proof of Dependent Choice, Compatible with Classical Logic. In *Logic in Computer Science*, pages 365–374, 2012.

[17] Pieter J. W. Hofstra. Partial Combinatory Algebras and Realizability Toposes. 2004.

[18] J. M. E. Hyland. The Effective Topos. In *The L. E. J. Brouwer Centenary Symposium*, volume 110 of *Studies in Logic and the Foundations of Mathematics*, pages 165–216. North-Holland, 1982.

[19] J. M. E. Hyland, P. T. Johnstone, and A. M. Pitts. Tripos Theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 88(2):205–232, 1980.

[20] Bart Jacobs. *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1998.

[21] Bart Jacobs. Personal communication, 2014.

[22] R. B. Jensen. Independence of the axiom of dependent choices from the countable axiom of choice (abstract). *Journal of Symbolic Logic*, 31(2):294, 1966.

[23] Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*, volume 1. Oxford University Press, 2002.

[24] S. C. Kleene. On the Interpretation of Intuitionistic Number Theory. *The Journal of Symbolic Logic*, 10(4):109–124, 1945.

[25] Stephen Cole Kleene and Richard Eugene Vesley. *The Foundations of Intuitionistic Mathematics: Especially in Relation to Recursive Functions*, volume 39 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1965.

[26] Saul A. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16(16):83–94, 1963.

[27] Robert S. Lubarsky. On the Cauchy Completeness of the Constructive Cauchy Reals. *Electronic Notes in Theoretical Computer Science*, 167:225–254, 2007. Proceedings of the Third International Conference on Computability and Complexity in Analysis.

[28] A. A. Markov. On the Continuity of Constructive Functions. *Uspekhi Matematicheskikh Nauk*, 9(3):226–230, 1954. Meetings of the Moscow Mathematical Society.

[29] Donald Michie. "Memo" Functions and Machine Learning. *Nature*, 218(5136):19–22, 1968.

[30] Étienne Miquey. A Sequent Calculus with Dependent Types for Classical Arithmetic. In *Logic in Computer Science*, pages 720–729, 2018.

[31] Eugenio Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[32] Joan Rand Moschovakis and Garyfallia Vafeiadou. Some Axioms for Constructive Analysis. *Archive for Mathematical Logic*, 51(5-6):443–459, 2012.

[33] John Myhill. Constructive Set Theory. *The Journal of Symbolic Logic*, 40(3):347–382, 1975.

[34] Gerhard Osius. Categorical Set Theory: A Characterization of the Category of Sets. *Journal of Pure and Applied Algebra*, 4(1):79–119, 1974.

[35] Andrew M. Pitts. Tripos Theory in Retrospect. *Mathematical Structures in Computer Science*, 12(3):265–279, 2002.

[36] Michael Rathjen. Choice Principles in Constructive and Classical Set Theories. In *Logic Colloquium*, Lecture Notes in Logic, pages 299–326. Cambridge University Press, 2002.

[37] Fred Richman. Constructive Mathematics without Choice. In *Reuniting the Antipodes — Constructive and Nonstandard Views of the Continuum*, pages 199–205. Springer Netherlands, 2001.

[38] Fred Richman. Pointwise Differentiability. In *Reuniting the Antipodes — Constructive and Nonstandard Views of the Continuum*, pages 207–210. Springer Netherlands, 2001.

[39] Wim Ruitenburg. Constructing Roots of Polynomials over the Complex Numbers. In *Computational Aspects of Lie Group Representations and Related Topics*, pages 107–128, 1990.

[40] Peter M. Schuster. Elementary Choiceless Constructive Analysis. In *Computer Science Logic*, pages 512–526, 2000.

[41] Peter M. Schuster. Countable Choice as a Questionable Uniformity Principle. *Philosophia Mathematica*, 12(2):106–134, 2004.

[42] Dana Scott. Completeness and Axiomatizability in Many-Valued Logic. In *Proceedings of the Tarski Symposium*, pages 411–436, 1974.

[43] A. S. Troelstra. History of Constructivism in the 20th Century. In *Set Theory, Arithmetic, and Foundations of Mathematics: Theorems, Philosophies*, Lecture Notes in Logic, pages 150–179. Cambridge University Press, 2011.

[44] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction, Volume I*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1988.

[45] Benno van den Berg. *Predicative Topos Theory and Models for Constructive Set Theory*. PhD thesis, Utrecht University, 2006.

[46] Jean van Heijenoort. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, 1967.

[47] Jaap van Oosten. *Realizability: An Introduction to its Categorical Side*, volume 152 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 2008.

Coq Formalization: Table of Contents

A Overview

The contributions of this paper have all been mechanically verified. This appendix is intended to let the reader know what exactly has been mechanically verified. In particular, it provides all the relevant formal definitions and formal statements of the lemmas and theorems. It does not, however, walk through the proofs in detail. For that, we refer the interested reader to directly interact with the Coq proofs themselves [13].

A-A. Conventions

Because Coq does not have quotient types and true subset types (due to the lack of baked-in proof irrelevance with support for unification), we employ a convention of define a type, then a validity predicate on that type, and then an equivalence relation on that type if appropriate. We only use elements of a type that are valid.

A-B. Metatheoretic Assumptions

The proof only makes one metatheoretic assumption, countable choice, in two places: the proof that all PCAs model countable choice, and the proof that Mem-SCA models countable choice.

B Categories

B-A. Common.v

We use **Set** to ensure that a type belongs to the universe of sets rather than some larger universe. We cannot use the standard Set universe because we need **Set** to contain Prop to model impredicativity.

Definition **Set** : Type := Type.

Inductive $\varnothing$: **Set** :=.

B-B. Categories.v

B-B1. Definition of a Category

This definition is standard. We include here only the structural components so that the reader may be introduced the notation.

Module Type *Category*.

Parameter $\mathcal{O}$: Type.
Parameter $\swarrow_{\mathcal{O}}$: $\mathcal{O} \to$ Prop.

Parameter $\rightsquigarrow$: $\mathcal{O} \to \mathcal{O} \to$ Type.
Parameter $\swarrow_{\rightsquigarrow}$: $\forall\ \{o_1\ o_2 : \mathcal{O}\}, \rightsquigarrow o_1\ o_2 \to$ Prop.
Parameter $\approx_{\rightsquigarrow}$: $\forall\ \{o_1\ o_2 : \mathcal{O}\}, \rightsquigarrow o_1\ o_2 \to \rightsquigarrow o_1\ o_2 \to$ Prop.

End *Category*.

B-C. Cartesian.v

B-C1. Definition of a Cartesian Category

This definition is standard. We include here only the structural components so that the reader may be introduced the notation.

Module Type *CartesianCategory*.
Include *Category*.

Parameter $\mathbf{1}$: $\mathcal{O}$.
Parameter ! : $\forall\ o : \mathcal{O}, \rightsquigarrow o\ \mathbf{1}$.

Parameter $\times$: $\mathcal{O} \to \mathcal{O} \to \mathcal{O}$.
Parameter $\langle \cdot, \cdot \rangle$: $\forall\ \{o\ o_1\ o_2 : \mathcal{O}\}, \rightsquigarrow o\ o_1 \to \rightsquigarrow o\ o_2 \to \rightsquigarrow o\ (\times\ o_1\ o_2)$.
Parameter π_1 : $\forall\ o_1\ o_2 : \mathcal{O}, \rightsquigarrow (\times\ o_1\ o_2)\ o_1$.
Parameter π_2 : $\forall\ o_1\ o_2 : \mathcal{O}, \rightsquigarrow (\times\ o_1\ o_2)\ o_2$.

End *CartesianCategory*.

B-C2. Definition of a Cartesian-Closed Category

This definition is standard. We include here only the structural components so that the reader may be introduced the notation.

Module Type *CartesianClosedCategory*.
Include *CartesianCategory*.

Parameter $\Rightarrow$: $\mathcal{O} \to \mathcal{O} \to \mathcal{O}$.
Parameter Λ : $\forall\ \{o\ o_1\ o_2 : \mathcal{O}\}, \rightsquigarrow (\times\ o\ o_1)\ o_2 \to \rightsquigarrow o\ (\Rightarrow o_1\ o_2)$.
Parameter ***eval*** : $\forall\ o_1\ o_2 : \mathcal{O}, \rightsquigarrow (\times\ (\Rightarrow o_1\ o_2)\ o_1)\ o_2$.

End *CartesianClosedCategory*.

B-C3. Definition of a Nautral-Number Object

This definition is standard. We include here only the structural components so that the reader may be introduced the notation.

Module Type *NaturalNumberObject* (*CC* : *CartesianCategory*).

Parameter $\boldsymbol{N}$: $\mathcal{O}$.

Parameter $\boldsymbol{Z}$: $\rightsquigarrow \mathbf{1}\ \boldsymbol{N}$.

```
Parameter S : ⇝ N N.
Parameter rec_N : ∀ {o : 𝒪}, ⇝ 1 o → ⇝ o o → ⇝ N o.
End NaturalNumberObject.
```

B-D. Sets.v

B-D1. Definition of the Category of Inhabited Sets

```
Module InhabitedSets <: CartesianClosedCategory.
  Record 𝒪 : Type := { set : Set; ✓set : set → Prop; ≈set : set → set → Prop }.
```

Objects are required to be *inhabited* sets (but not pointed sets), as indicated by *sinh*.

```
  Record ✓𝒪 (o : 𝒪) : Prop
  := { sinh : ∃ s : set o, ✓set o s
     ; srefl : ∀ s : set o, ✓set o s → ≈set o s s
     ; ssym : ∀ s s' : set o, ✓set o s → ✓set o s' → ≈set o s s' → ≈set o s' s
     ; strans : ∀ s s' s'' : set o, ✓set o s → ✓set o s' → ✓set o s'' → ≈set o s s' → ≈set o s' s'' → ≈set o s s'' }.
  Definition ⇝ (o1 o2 : 𝒪) : Type := set o1 → set o2.
```

Note that morphisms are not required to preserve the required inhabitant, making this the category of inhabited sets rather than pointed sets.

```
  Record ✓⇝ (o1 o2 : 𝒪) (m : ⇝ o1 o2) : Prop
  := { mpresv : ∀ s1 : set o1, ✓set o1 s1 → ✓set o2 (m s1)
     ; mprese : ∀ s1 s1' : set o1, ✓set o1 s1 → ✓set o1 s1' → ≈set o1 s1 s1' → ≈set o2 (m s1) (m s1') }.
  Definition ≈⇝ {o1 o2 : 𝒪} (m1 m2 : ⇝ o1 o2) : Prop := ∀ s1 s1' : set o1, ✓set o1 s1 → ✓set o1 s1' → ≈set o1 s1 s1' →
≈set o2 (m1 s1) (m2 s1').
```

We omit the remainder of this module as it simply demonstrates that inhabited sets form a cartesian-closed category, which is not novel. Note, though, that the ability to customize the validity predicate and equivalence relation are necessary for this construction.

```
End InhabitedSets.
Module InhabitedSetsNat <: NaturalNumberObject InhabitedSets.
```

We omit the contents of this module as they simply demonstrate that inhabited sets have the obvious natural-number object.

```
End InhabitedSetsNat.
```

C Combinatory Algebras

C-A. Expressions.v

C-A1. Definition of Applicative Expressions

```
Module ApplicativeExpression.
  Fixpoint V (C : Set) (n : ℕ) : Type := match n with 0 ↦ C | S n ↦ option (V C n) end.
  Fixpoint vcode {C : Type} (c : C) (n : ℕ) : V C n := match n with 0 ↦ c | S n ↦ Some (vcode c n) end.
  Inductive E? {C : Set} {n : ℕ} : Type := evar (c : V C n) | · (e_f e_a : E?).
  Definition Expr (C : Set) : Type := E? C 0.
  Fixpoint ✓V {C : Set} (✓C : C → Prop) {n : ℕ} : V C n → Prop
  := match n with 0 ↦ ✓C | S n ↦ λ v ↦ match v with None ↦ True | Some v ↦ ✓V ✓C v end end.
  Inductive ✓E {C : Set} (✓C : C → Prop) {n : ℕ} : E? C n → Prop
  := evarv (c : V C n) : ✓V ✓C c → ✓E ✓C (evar n c)
   | eappv (e_f e_a : E? C n) : ✓E ✓C e_f → ✓E ✓C e_a → ✓E ✓C (· e_f e_a).
  Fixpoint ė[·] {C : Set} (c : C) {n : ℕ} (e : E? C (S n)) : E? C n
  := match e with evar _ v ↦ evar n (match v with None ↦ vcode c n | Some v ↦ v end) | · e_f e_a ↦ · (ė[·] c e_f) (ė[·] c
e_a) end.
  Lemma esubstv (C : Set) (✓C : C → Prop) (c : C) (n : ℕ) (e : E? C (S n)) : ✓C c → ✓E ✓C e → ✓E ✓C (ė[·] c e).
  Definition (·) {C : Set} (c : C) {n : ℕ} : E? C n → E? C n := evar n (vcode c n).
  Lemma ecodev (C : Set) (✓C : C → Prop) (c : C) (n : ℕ) : ✓C c → ✓E ✓C (n := n) ((·) c).
End ApplicativeExpression.
```

C-B. RCAs.v

C-B1. Definition of a Relational Applicative Structure

This is the formal statement of Definition 4.1.

```
Module Type RelationalApplicativeStructure.
  Parameter C : Set.
  Parameter ✓C : C → Prop.
  Parameter ↓c : C → C → C → Prop.
  Parameter ↓ : C → C → Prop.
  Parameter preservation : ∀ c_f c_a c_r : C, ✓C c_f → ✓C c_a → ↓c c_f c_a c_r → ✓C c_r.
  Parameter progress : ∀ c_f c_a : C, ✓C c_f → ✓C c_a → ↓ c_f c_a → ∃ c_r : C, ↓c c_f c_a c_r.
End RelationalApplicativeStructure.
```

C-B2. Definition of Relational Reduction and Termination of Applicative Expressions

```
Module RelationalApplicativeExpression.
  Inductive ↓c^E {C : Set} (↓c : C → C → C → Prop) : E0 C → C → Prop
  := revar (c : C) : ↓c^E ↓c (evar 0 c) c
   | reapp (e_f e_a : E0 C) (c_f c_a c_r : C) : ↓c^E ↓c e_f c_f → ↓c^E ↓c e_a c_a → ↓c c_f c_a c_r → ↓c^E ↓c (· e_f e_a) c_r.
```

Inductive $\downarrow^E$ {C : **Set**} ($\downarrow_c$: $C \to C \to C \to$ Prop) ($\downarrow$: $C \to C \to$ Prop) : $E_0\ C \to$ Prop
:= *tevar* (c : C) : $\downarrow^E \downarrow_c \downarrow$ (*evar* 0 c)
| *teapp* (e_f e_a : $E_0\ C$) : $\downarrow^E \downarrow_c \downarrow e_f \to (\forall\ c_f : C, \downarrow_c^E \downarrow_c e_f\ c_f \to \downarrow^E \downarrow_c \downarrow e_a \wedge (\forall\ c_a : C, \downarrow_c^E \downarrow_c e_a\ c_a \to \downarrow c_f\ c_a)) \to \downarrow^E \downarrow_c \downarrow (\cdot\ e_f\ e_a)$.
Definition $\downarrow_\phi$ {C : **Set**} ($\downarrow_c$: $C \to C \to C \to$ Prop) ($\downarrow$: $C \to C \to$ Prop) ($c_f\ c_a$: C) (ϕ_r : $C \to$ Prop) : Prop
:= $\downarrow c_f\ c_a \wedge (\forall\ c_r : C, \downarrow_c\ c_f\ c_a\ c_r \to \phi_r\ c_r)$.

Lemma *termred_forall* {C I : **Set**} ($\downarrow_c$: $C \to C \to C \to$ Prop) ($\downarrow$: $C \to C \to$ Prop) ($\swarrow_I$: $I \to$ Prop) ($c_f\ c_a$: C) (ϕ_r : $I \to C \to$ Prop) : $(\exists\ i : I, \swarrow_I\ i) \to (\forall\ i : I, \swarrow_I\ i \to \downarrow_\phi \downarrow_c \downarrow c_f\ c_a\ (\phi_r\ i)) \to \downarrow_\phi \downarrow_c \downarrow c_f\ c_a\ (\lambda\, c_r \mapsto \forall\ i : I, \swarrow_I\ i \to \phi_r\ i\ c_r)$.

Fixpoint $\downarrow_\phi^E$ {C : **Set**} ($\downarrow_c$: $C \to C \to C \to$ Prop) ($\downarrow$: $C \to C \to$ Prop) (e : $E_0\ C$) (ϕ_r : $C \to$ Prop) : Prop
:= match e with *evar* _ $c \mapsto \phi_r\ c$ | $\cdot\ e_f\ e_a \mapsto \downarrow_\phi^E \downarrow_c \downarrow e_f\ (\lambda\, c_f \mapsto \downarrow_\phi^E \downarrow_c \downarrow e_a\ (\lambda\, c_a \mapsto \downarrow_\phi \downarrow_c \downarrow c_f\ c_a\ \phi_r))$ end.

Lemma *termredexpr* {C : **Set**} ($\downarrow_c$: $C \to C \to C \to$ Prop) ($\downarrow$: $C \to C \to$ Prop) (e : $E_0\ C$) (ϕ_r : $C \to$ Prop) : $\downarrow_\phi^E \downarrow_c \downarrow e\ \phi_r \to \downarrow^E \downarrow_c \downarrow e \wedge (\forall\ c_r : C, \downarrow_c^E \downarrow_c e\ c_r \to \phi_r\ c_r)$.

End *RelationalApplicativeExpression*.

C-B3. Definition of a Relational Combinatory Algebra

This is the formal statement of Definition 4.2.

Module Type *RelationalCombinatoryAlgebra*.
Include *RelationalApplicativeStructure*.

Parameter c_λ : $\forall\ n : \mathbb{N}, E_?\ C\ (S\ n) \to C$.
Parameter *cencodev* : $\forall\ n : \mathbb{N}, \forall\ e : E_?\ C\ (S\ n), \swarrow_E \swarrow_C e \to \swarrow_C (c_\lambda\ n\ e)$.
Parameter *red_encode_S* : $\forall\ n : \mathbb{N}, \forall\ e : E_?\ C\ (S\ (S\ n)), \forall\ c_a\ c_r : C, \swarrow_E \swarrow_C e \to \swarrow_C c_a \to \downarrow_c (c_\lambda\ (S\ n)\ e)\ c_a\ c_r \to c_\lambda\ n\ (\dot{e}[\cdot]\ c_a\ e) = c_r$.
Parameter *red_encode_0* : $\forall\ e : E_?\ C\ 1, \forall\ c_a\ c_r : C, \swarrow_E \swarrow_C e \to \swarrow_C c_a \to \downarrow_c (c_\lambda\ 0\ e)\ c_a\ c_r \to \downarrow_c^E \downarrow_c (\dot{e}[\cdot]\ c_a\ e)\ c_r$.
Parameter *term_encode_S* : $\forall\ n : \mathbb{N}, \forall\ e : E_?\ C\ (S\ (S\ n)), \forall\ c_a : C, \swarrow_E \swarrow_C e \to \swarrow_C c_a \to \downarrow (c_\lambda\ (S\ n)\ e)\ c_a$.
Parameter *term_encode_0* : $\forall\ e : E_?\ C\ 1, \forall\ c_a : C, \swarrow_E \swarrow_C e \to \swarrow_C c_a \to \downarrow^E \downarrow_c \downarrow (\dot{e}[\cdot]\ c_a\ e) \to \downarrow (c_\lambda\ 0\ e)\ c_a$.

End *RelationalCombinatoryAlgebra*.

C-C. PCAs.v

C-C1. Definition of a Partial Combinatory Algebra

This is the formal statement of Definition 3.2

Module Type *PartialCombinatoryAlgebra*.
Include *RelationalCombinatoryAlgebra*.

Parameter *red_deterministic* : $\forall\ c_f\ c_a\ c_r\ c_r', \swarrow_C c_f \to \swarrow_C c_a \to \downarrow_c c_f\ c_a\ c_r \to \downarrow_c c_f\ c_a\ c_r' \to c_r = c_r'$.
Parameter *red_term* : $\forall\ c_f\ c_a\ c_r : C, \swarrow_C c_f \to \swarrow_C c_a \to \swarrow_C c_r \to \downarrow_c c_f\ c_a\ c_r \to \downarrow c_f\ c_a$.

End *PartialCombinatoryAlgebra*.

C-D. SCAs.v

C-D1. Definition of a Stateful Applicative Structure

This is the formal statement of Definition 5.1. One difference, though, is that in this definition we allow code-validity to depend on the current state so long as code-validity is future-stable. Although our proofs do not rely on this additional degree of flexibility, it can be convenient for keeping the model clean, say by guaranteeing that any references a valid code has to the state necessarily refer to allocated locations.

Module Type *StatefulApplicativeStructure*.

Parameter Σ : **Set**.
Parameter $\swarrow_\Sigma$: $\Sigma \to$ Prop.
Parameter *sinhabited* : $\exists\ \sigma : \Sigma, \swarrow_\Sigma \sigma$.

Parameter $\leq$: $\Sigma \to \Sigma \to$ Prop.
Parameter *frefl* : $\forall\ \sigma : \Sigma, \swarrow_\Sigma \sigma \to \leq \sigma\ \sigma$.
Parameter *ftrans* : $\forall\ \sigma\ \sigma'\ \sigma'' : \Sigma, \swarrow_\Sigma \sigma \to \swarrow_\Sigma \sigma' \to \swarrow_\Sigma \sigma'' \to \leq \sigma\ \sigma' \to \leq \sigma'\ \sigma'' \to \leq \sigma\ \sigma''$.

Parameter C : **Set**.

Parameter $\swarrow_C$: $\Sigma \to C \to$ Prop.
Parameter *codev_fut* : $\forall\ \sigma\ \sigma' : \Sigma, \forall\ c : C, \swarrow_\Sigma \sigma \to \swarrow_\Sigma \sigma' \to \leq \sigma\ \sigma' \to \swarrow_C \sigma\ c \to \swarrow_C \sigma'\ c$.

Parameter $\downarrow_c$: $\Sigma \to C \to C \to \Sigma \to C \to$ Prop.
Parameter $\downarrow$: $\Sigma \to C \to C \to$ Prop.

Parameter *preservation* : $\forall\ \sigma : \Sigma, \forall\ c_f\ c_a : C, \forall\ \sigma' : \Sigma, \forall\ c_r : C, \swarrow_\Sigma \sigma \to \swarrow_C \sigma\ c_f \to \swarrow_C \sigma\ c_a \to \downarrow_c \sigma\ c_f\ c_a\ \sigma'\ c_r \to \swarrow_\Sigma \sigma' \wedge \leq \sigma\ \sigma' \wedge \swarrow_C \sigma'\ c_r$.
Parameter *progress* : $\forall\ \sigma : \Sigma, \forall\ c_f\ c_a : C, \swarrow_\Sigma \sigma \to \swarrow_C \sigma\ c_f \to \swarrow_C \sigma\ c_a \to \downarrow \sigma\ c_f\ c_a \to \exists\ \sigma' : \Sigma, \exists\ c_r : C, \downarrow_c \sigma\ c_f\ c_a\ \sigma'\ c_r$.

End *StatefulApplicativeStructure*.

C-D2. Definition of Stateful Reduction and Termination of Applicative Expressions

Module *StatefulApplicativeExpression*.

Inductive $\downarrow_c^E$ {Σ C : **Set**} ($\downarrow_c$: $\Sigma \to C \to C \to \Sigma \to C \to$ Prop) : $\Sigma \to E_0\ C \to \Sigma \to C \to$ Prop
:= *revar* (σ : Σ) (c : C) : $\downarrow_c^E \downarrow_c \sigma$ (*evar* 0 c) σ c
| *reapp* (σ : Σ) (e_f e_a : $E_0\ C$) (σ' : Σ) (c_f : C) (σ'' : Σ) (c_a : C) (s''' : Σ) (c_r : C) : $\downarrow_c^E \downarrow_c \sigma\ e_f\ \sigma'\ c_f \to \downarrow_c^E \downarrow_c \sigma'\ e_a\ \sigma''\ c_a \to \downarrow_c \sigma''\ c_f\ c_a\ s'''\ c_r \to \downarrow_c^E \downarrow_c \sigma\ (\cdot\ e_f\ e_a)\ s'''\ c_r$.
Inductive $\downarrow^E$ {Σ C : **Set**} ($\downarrow_c$: $\Sigma \to C \to C \to \Sigma \to C \to$ Prop) ($\downarrow$: $\Sigma \to C \to C \to$ Prop) : $\Sigma \to E_0\ C \to$ Prop
:= *tevar* (σ : Σ) (c : C) : $\downarrow^E \downarrow_c \downarrow \sigma$ (*evar* 0 c)
| *teapp* (σ : Σ) (e_f e_a : $E_0\ C$) : $\downarrow^E \downarrow_c \downarrow \sigma\ e_f \to (\forall\ \sigma' : \Sigma, \forall\ c_f : C, \downarrow_c^E \downarrow_c \sigma\ e_f\ \sigma'\ c_f \to \downarrow^E \downarrow_c \downarrow \sigma'\ e_a \wedge (\forall\ \sigma'' : \Sigma, \forall\ c_a : C, \downarrow_c^E \downarrow_c \sigma'\ e_a\ \sigma''\ c_a \to \downarrow \sigma''\ c_f\ c_a)) \to \downarrow^E \downarrow_c \downarrow \sigma\ (\cdot\ e_f\ e_a)$.
Definition $\downarrow_\phi$ {Σ C : **Set**} ($\downarrow_c$: $\Sigma \to C \to C \to \Sigma \to C \to$ Prop) ($\downarrow$: $\Sigma \to C \to C \to$ Prop) (σ : Σ) ($c_f\ c_a$: C) (ϕ_r : $\Sigma \to C \to$ Prop) : Prop

$:= \downarrow\ \sigma\ c_f\ c_a \wedge (\forall\ \sigma' : \Sigma, \forall\ c_r : C, \downarrow_c\ \sigma\ c_f\ c_a\ \sigma'\ c_r \to \phi_r\ \sigma'\ c_r)$.

Lemma *termred_forall* $\{\Sigma\ C\ I : \mathbf{Set}\}$ $(\downarrow_c : \Sigma \to C \to C \to \Sigma \to C \to \text{Prop})$ $(\downarrow : \Sigma \to C \to C \to \text{Prop})$ $(\checkmark_I : I \to \text{Prop})$ $(\sigma : \Sigma)$ $(c_f\ c_a : C)$ $(\phi_r : I \to \Sigma \to C \to \text{Prop})$: $(\exists\ i : I, \checkmark_I\ i) \to (\forall\ i : I, \checkmark_I\ i \to \downarrow_\phi \downarrow_c \downarrow\ \sigma\ c_f\ c_a\ (\phi_r\ i)) \to \downarrow_\phi \downarrow_c \downarrow\ \sigma\ c_f\ c_a\ (\lambda\ \sigma'\ c_r \mapsto \forall\ i : I, \checkmark_I\ i \to \phi_r\ i\ \sigma'\ c_r)$.

Fixpoint $\downarrow_\phi^E$ $\{\Sigma\ C : \mathbf{Set}\}$ $(\downarrow_c : \Sigma \to C \to C \to \Sigma \to C \to \text{Prop})$ $(\downarrow : \Sigma \to C \to C \to \text{Prop})$ $(\sigma : \Sigma)$ $(e : E_0\ C)$ $(\phi_r : \Sigma \to C \to \text{Prop})$: Prop
:= match e with *evar* _ $c \mapsto \phi_r\ \sigma\ c \mid \cdot\ e_f\ e_a \mapsto \downarrow_\phi^E \downarrow_c \downarrow\ \sigma\ e_f\ (\lambda\ \sigma'\ c_f \mapsto \downarrow_\phi^E \downarrow_c \downarrow\ \sigma'\ e_a\ (\lambda\ \sigma''\ c_a \mapsto \downarrow_\phi \downarrow_c \downarrow\ \sigma''\ c_f\ c_a\ \phi_r))$ end.
Lemma *termredexpr* $\{\Sigma\ C : \mathbf{Set}\}$ $(\downarrow_c : \Sigma \to C \to C \to \Sigma \to C \to \text{Prop})$ $(\downarrow : \Sigma \to C \to C \to \text{Prop})$ $(\sigma : \Sigma)$ $(e : E_0\ C)$ $(\phi_r : \Sigma \to C \to \text{Prop})$: $\downarrow_\phi^E \downarrow_c \downarrow\ \sigma\ e\ \phi_r \to \downarrow^E \downarrow_c \downarrow\ \sigma\ e \wedge (\forall\ \sigma' : \Sigma, \forall\ c_r : C, \downarrow_c^E \downarrow_c\ \sigma\ e\ \sigma'\ c_r \to \phi_r\ \sigma'\ c_r)$.

End *StatefulApplicativeExpression*.

C-D3. Definition of a Stateful Combinatory Algebra

This is the formal statement of Definition 5.2.

Module Type *StatefulCombinatoryAlgebra*.
Include *StatefulApplicativeStructure*.

Parameter $c_\lambda : \forall\ n : \mathbb{N}, E_?\ C\ (S\ n) \to C$.
Parameter *cencodev* : $\forall\ n : \mathbb{N}, \forall\ e : E_?\ C\ (S\ n), \forall\ \sigma : \Sigma, \checkmark_\Sigma\ \sigma \to \checkmark_E\ (\checkmark_C\ \sigma)\ e \to \checkmark_C\ \sigma\ (c_\lambda\ n\ e)$.
Parameter *red_encode_S* : $\forall\ \sigma\ \sigma' : \Sigma, \forall\ n : \mathbb{N}, \forall\ e : E_?\ C\ (S\ (S\ n)), \forall\ c_a\ c_r : C, \checkmark_\Sigma\ \sigma \to \checkmark_E\ (\checkmark_C\ \sigma)\ e \to \checkmark_C\ \sigma\ c_a \to \downarrow_c \sigma\ (c_\lambda\ (S\ n)\ e)\ c_a\ \sigma'\ c_r \to \sigma' = \sigma \wedge c_\lambda\ n\ (\dot{e}[\cdot]\ c_a\ e) = c_r$.
Parameter *red_encode_0* : $\forall\ \sigma\ \sigma' : \Sigma, \forall\ e : E_?\ C\ 1, \forall\ c_a\ c_r : C, \checkmark_\Sigma\ \sigma \to \checkmark_E\ (\checkmark_C\ \sigma)\ e \to \checkmark_C\ \sigma\ c_a \to \downarrow_c \sigma\ (c_\lambda\ 0\ e)\ c_a\ \sigma'\ c_r \to \downarrow_c^E \downarrow_c \sigma\ (\dot{e}[\cdot]\ c_a\ e)\ \sigma'\ c_r$.
Parameter *term_encode_S* : $\forall\ \sigma : \Sigma, \forall\ n : \mathbb{N}, \forall\ e : E_?\ C\ (S\ (S\ n)), \forall\ c_a : C, \checkmark_\Sigma\ \sigma \to \checkmark_E\ (\checkmark_C\ \sigma)\ e \to \checkmark_C\ \sigma\ c_a \to \downarrow\ \sigma\ (c_\lambda\ (S\ n)\ e)\ c_a$.
Parameter *term_encode_0* : $\forall\ \sigma : \Sigma, \forall\ e : E_?\ C\ 1, \forall\ c_a : C, \checkmark_\Sigma\ \sigma \to \checkmark_E\ (\checkmark_C\ \sigma)\ e \to \checkmark_C\ \sigma\ c_a \to \downarrow^E \downarrow_c \downarrow\ \sigma\ (\dot{e}[\cdot]\ c_a\ e) \to \downarrow\ \sigma\ (c_\lambda\ 0\ e)\ c_a$.

End *StatefulCombinatoryAlgebra*.

C-E. Lambdas.v

C-E1. Framework for Building Codes with Lambda-Terms and Primitives

Module *LambdaTerm*.

Defines λ-body expressions L parameterized by a set of codes C, and defines codes $C_?$ parameterized by a set of primitives P such that a code is either a primitive or a λ-code of a λ-body expression.

End *LambdaTerm*.

C-F. FreeRCA.v

C-F1. Framework for Building Relational Combinatory Algebras with Lambda-Terms and Primitives

We elide the construction as it is just tedious and made complex by the fact that there is no direct way to define mutually dependent inductive types or propositions across modules. We only show the module type for specifying the set of primitives and their termination and reduction behavior.

Module *FreeRelationalCombinatoryCode*.

Defines relational application $\downarrow_c^\lambda$ and termination $\downarrow^\lambda$ for λ-body expressions, and application $\downarrow_c^{C_?}$ and termination $\downarrow^{C_?}$ for codes, each parameterized by application and termination rules for the appropriate unknown sets of codes/primitives.

End *FreeRelationalCombinatoryCode*.

Module Type *PrimitiveApplicativeStructure*.

Parameter P : **Set**.
Parameter $\checkmark_P : P \to \text{Prop}$.
Parameter $\downarrow_c^p : P \to C_?\ P \to C_?\ P \to \text{Prop}$.
Parameter $\downarrow^p : P \to C_?\ P \to \text{Prop}$.
Parameter *preservation_prim* : $\forall\ p_f : P, \forall\ c_a : C_?\ P, \forall\ c_r : C_?\ P, \checkmark_P\ p_f \to \checkmark_{C_?}\ \checkmark_P\ c_a \to \downarrow_c^p\ p_f\ c_a\ c_r \to \checkmark_{C_?}\ \checkmark_P\ c_r$.
Parameter *progress_prim* : $\forall\ p_f : P, \forall\ c_a : C_?\ P, \checkmark_P\ p_f \to \checkmark_{C_?}\ \checkmark_P\ c_a \to \downarrow^p\ p_f\ c_a \to \exists\ c_r : C_?\ P, \downarrow_c^p\ p_f\ c_a\ c_r$.

End *PrimitiveApplicativeStructure*.

Module *FreeRelationalCombinatoryAlgebra* (*PAS* : *PrimitiveApplicativeStructure*) <: *RelationalCombinatoryAlgebra*.
Include *FreeRelationalCombinatoryCode*.
Include *PAS*.

Tediously ties the mutually recursive knot.

End *FreeRelationalCombinatoryAlgebra*.

C-G. FreeSCA.v

C-G1. Framework for Building Stateful Combinatory Algebras with Lambda-Terms and Primitives

We elide the construction as it is just tedious and made complex by the fact that there is no direct way to define mutually dependent inductive types or propositions across modules. We only show the module type for specifying the set of primitives and their termination and reduction behavior.

Module *FreeStatefulCombinatoryCode*.

Defines stateful application $\downarrow_c^\lambda$ and termination $\downarrow^\lambda$ for λ-body expressions, and application $\downarrow_c^{C_?}$ and termination $\downarrow^{C_?}$ for codes, each parameterized by application and termination rules for the appropriate unknown sets of codes/primitives and states.

End *FreeStatefulCombinatoryCode*.

Module Type *PrimitiveApplicativeStructure*.

Parameter Σ : **Set**.
Parameter P : **Set**.

Parameter $\checkmark_\Sigma$: $\Sigma \to$ Prop.
Parameter *sinhabited* : $\exists\ \sigma : \Sigma,\ \checkmark_\Sigma\ \sigma$.

Parameter $\leq$: $\Sigma \to \Sigma \to$ Prop.
Parameter *frefl* : $\forall\ \sigma : \Sigma,\ \checkmark_\Sigma\ \sigma \to \leq\ \sigma\ \sigma$.
Parameter *ftrans* : $\forall\ \sigma\ \sigma'\ \sigma'' : \Sigma,\ \checkmark_\Sigma\ \sigma \to \checkmark_\Sigma\ \sigma' \to \checkmark_\Sigma\ \sigma'' \to \leq\ \sigma\ \sigma' \to \leq\ \sigma'\ \sigma'' \to \leq\ \sigma\ \sigma''$.

Parameter $\checkmark_P$: $\Sigma \to P \to$ Prop.
Parameter *primv_fut* : $\forall\ \sigma\ \sigma' : \Sigma,\ \forall\ p : P,\ \checkmark_\Sigma\ \sigma \to \checkmark_\Sigma\ \sigma' \to \leq\ \sigma\ \sigma' \to \checkmark_P\ \sigma\ p \to \checkmark_P\ \sigma'\ p$.

Parameter $\downarrow^p_c$: $\Sigma \to P \to C_?\ P \to \Sigma \to C_?\ P \to$ Prop.
Parameter $\downarrow^p$: $\Sigma \to P \to C_?\ P \to$ Prop.

Parameter *preservation_prim* : $\forall\ \sigma : \Sigma,\ \forall\ p_f : P,\ \forall\ c_a : C_?\ P,\ \forall\ \sigma' : \Sigma,\ \forall\ c_r : C_?\ P,\ \checkmark_\Sigma\ \sigma \to \checkmark_P\ \sigma\ p_f \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_a \to \downarrow^p_c\ \sigma\ p_f\ c_a\ \sigma'\ c_r \to \checkmark_\Sigma\ \sigma' \wedge \leq\ \sigma\ \sigma' \wedge \checkmark_{C_?}\ (\checkmark_P\ \sigma')\ c_r$.

Parameter *progress_prim* : $\forall\ \sigma : \Sigma,\ \forall\ p_f : P,\ \forall\ c_a : C_?\ P,\ \checkmark_\Sigma\ \sigma \to \checkmark_P\ \sigma\ p_f \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_a \to \downarrow^p\ \sigma\ p_f\ c_a \to \exists\ \sigma' : \Sigma,\ \exists\ c_r : C_?\ P,\ \downarrow^p_c\ \sigma\ p_f\ c_a\ \sigma'\ c_r$.

End *PrimitiveApplicativeStructure*.

Module *FreeStatefulCombinatoryAlgebra* (*PAS* : *PrimitiveApplicativeStructure*) <: *StatefulCombinatoryAlgebra*.
Include *FreeStatefulCombinatoryCode*.
Include *PAS*.

Tediously ties the mutually recursive knot.

End *FreeStatefulCombinatoryAlgebra*.

D Higher-Order Fibrations

D-A. HOFs.v

D-A1. Definition of a Higher-Order Fibration

Module Type *HigherOrderFibration* (*CCC* : *CartesianClosedCategory*).

Propositions

Parameter Φ : $\mathcal{O} \to$ Type.
Parameter $\checkmark_\Phi$: $\forall\ \{o : \mathcal{O}\},\ \Phi\ o \to$ Prop.

Entailment

Parameter $\vdash$: $\forall\ \{o : \mathcal{O}\},\ \Phi\ o \to \Phi\ o \to$ Prop.

Parameter *refl* : $\forall\ o : \mathcal{O},\ \forall\ \phi : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi \to \vdash\ \phi\ \phi$.
Parameter *trans* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2\ \phi_3 : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \checkmark_\Phi\ \phi_3 \to \vdash\ \phi_1\ \phi_2 \to \vdash\ \phi_2\ \phi_3 \to \vdash\ \phi_1\ \phi_3$.

Substition

Parameter $\dot\phi[\cdot]$: $\forall\ \{o_1\ o_2 : \mathcal{O}\},\ \forall\ m : \rightsquigarrow\ o_1\ o_2,\ \Phi\ o_2 \to \Phi\ o_1$.
Parameter *substv* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_2,\ \forall\ \phi_2 : \Phi\ o_2,\ \checkmark_\mathcal{O}\ o_1 \to \checkmark_\mathcal{O}\ o_2 \to \swarrow_\rightsquigarrow\ m \to \checkmark_\Phi\ \phi_2 \to \checkmark_\Phi\ (\dot\phi[\cdot]\ m\ \phi_2)$.
Parameter *substi* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_2,\ \forall\ \phi_2\ \phi_2' : \Phi\ o_2,\ \checkmark_\mathcal{O}\ o_1 \to \checkmark_\mathcal{O}\ o_2 \to \swarrow_\rightsquigarrow\ m \to \checkmark_\Phi\ \phi_2 \to \checkmark_\Phi\ \phi_2' \to \vdash\ \phi_2\ \phi_2' \to \vdash\ (\dot\phi[\cdot]\ m\ \phi_2)\ (\dot\phi[\cdot]\ m\ \phi_2')$.
Parameter *subste* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ m_1\ m_2 : \rightsquigarrow\ o_1\ o_2,\ \forall\ \phi_2 : \Phi\ o_2,\ \checkmark_\mathcal{O}\ o_1 \to \checkmark_\mathcal{O}\ o_2 \to \swarrow_\rightsquigarrow\ m_1 \to \swarrow_\rightsquigarrow\ m_2 \to \checkmark_\Phi\ \phi_2 \to \approx_\rightsquigarrow\ m_1\ m_2 \to \vdash\ (\dot\phi[\cdot]\ m_1\ \phi_2)\ (\dot\phi[\cdot]\ m_2\ \phi_2) \wedge \vdash\ (\dot\phi[\cdot]\ m_2\ \phi_2)\ (\dot\phi[\cdot]\ m_1\ \phi_2)$.

Parameter *substid* : $\forall\ o : \mathcal{O},\ \forall\ \phi : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi \to \vdash\ \phi\ (\dot\phi[\cdot]\ (id\ o)\ \phi) \wedge \vdash\ (\dot\phi[\cdot]\ (id\ o)\ \phi)\ \phi$.
Parameter *substcomp* : $\forall\ o_1\ o_2\ o_3 : \mathcal{O},\ \forall\ m_{12} : \rightsquigarrow\ o_1\ o_2,\ \forall\ m_{23} : \rightsquigarrow\ o_2\ o_3,\ \forall\ \phi_3 : \Phi\ o_3,\ \checkmark_\mathcal{O}\ o_1 \to \checkmark_\mathcal{O}\ o_2 \to \checkmark_\mathcal{O}\ o_3 \to \swarrow_\rightsquigarrow\ m_{12} \to \swarrow_\rightsquigarrow\ m_{23} \to \checkmark_\Phi\ \phi_3 \to \vdash\ (\dot\phi[\cdot]\ m_{12}\ (\dot\phi[\cdot]\ m_{23}\ \phi_3))\ (\dot\phi[\cdot]\ (;\ m_{12}\ m_{23})\ \phi_3) \wedge \vdash\ (\dot\phi[\cdot]\ (;\ m_{12}\ m_{23})\ \phi_3)\ (\dot\phi[\cdot]\ m_{12}\ (\dot\phi[\cdot]\ m_{23}\ \phi_3))$.

True

Parameter $\top$: $\forall\ o : \mathcal{O},\ \Phi\ o$.
Parameter *topv* : $\forall\ o : \mathcal{O},\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ (\top\ o)$.
Parameter *topi* : $\forall\ o : \mathcal{O},\ \forall\ \phi : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi \to \vdash\ \phi\ (\top\ o)$.
Parameter *tops* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_2,\ \checkmark_\mathcal{O}\ o_1 \to \checkmark_\mathcal{O}\ o_2 \to \swarrow_\rightsquigarrow\ m \to \vdash\ (\top\ o_1)\ (\dot\phi[\cdot]\ m\ (\top\ o_2))$.

Conjunction

Parameter $\wedge$: $\forall\ \{o : \mathcal{O}\},\ \Phi\ o \to \Phi\ o \to \Phi\ o$.
Parameter *conjv* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \checkmark_\Phi\ (\wedge\ \phi_1\ \phi_2)$.
Parameter *conji* : $\forall\ o : \mathcal{O},\ \forall\ \phi\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \vdash\ \phi\ \phi_1 \to \vdash\ \phi\ \phi_2 \to \vdash\ \phi\ (\wedge\ \phi_1\ \phi_2)$.
Parameter *conje1* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \vdash\ (\wedge\ \phi_1\ \phi_2)\ \phi_1$.
Parameter *conje2* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \vdash\ (\wedge\ \phi_1\ \phi_2)\ \phi_2$.
Parameter *conjs* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_2,\ \forall\ \phi_1\ \phi_2 : \Phi\ o_2,\ \checkmark_\mathcal{O}\ o_1 \to \checkmark_\mathcal{O}\ o_2 \to \swarrow_\rightsquigarrow\ m \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \vdash\ (\wedge\ (\dot\phi[\cdot]\ m\ \phi_1)\ (\dot\phi[\cdot]\ m\ \phi_2))\ (\dot\phi[\cdot]\ m\ (\wedge\ \phi_1\ \phi_2))$.

False

Parameter $\bot$: $\forall\ o : \mathcal{O},\ \Phi\ o$.
Parameter *botv* : $\forall\ o : \mathcal{O},\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ (\bot\ o)$.
Parameter *bote* : $\forall\ o : \mathcal{O},\ \forall\ \phi : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi \to \vdash\ (\bot\ o)\ \phi$.
Parameter *bots* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_2,\ \checkmark_\mathcal{O}\ o_1 \to \checkmark_\mathcal{O}\ o_2 \to \swarrow_\rightsquigarrow\ m \to \vdash\ (\dot\phi[\cdot]\ m\ (\bot\ o_2))\ (\bot\ o_1)$.

Disjunction

Parameter $\vee$: $\forall\ \{o : \mathcal{O}\},\ \Phi\ o \to \Phi\ o \to \Phi\ o$.
Parameter *disjv* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \checkmark_\Phi\ (\vee\ \phi_1\ \phi_2)$.
Parameter *disji1* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \vdash\ \phi_1\ (\vee\ \phi_1\ \phi_2)$.
Parameter *disji2* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \vdash\ \phi_2\ (\vee\ \phi_1\ \phi_2)$.
Parameter *disje* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2\ \phi : \Phi\ o,\ \checkmark_\mathcal{O}\ o \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \checkmark_\Phi\ \phi \to \vdash\ \phi_1\ \phi \to \vdash\ \phi_2\ \phi \to \vdash\ (\vee\ \phi_1\ \phi_2)\ \phi$.
Parameter *disjs* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_2,\ \forall\ \phi_1\ \phi_2 : \Phi\ o_2,\ \checkmark_\mathcal{O}\ o_1 \to \checkmark_\mathcal{O}\ o_2 \to \swarrow_\rightsquigarrow\ m \to \checkmark_\Phi\ \phi_1 \to \checkmark_\Phi\ \phi_2 \to \vdash\ (\dot\phi[\cdot]\ m\ (\vee\ \phi_1\ \phi_2))\ (\vee\ (\dot\phi[\cdot]\ m\ \phi_1)\ (\dot\phi[\cdot]\ m\ \phi_2))$.

Implication

PARAMETER $\supset : \forall\ \{o : \mathcal{O}\},\ \Phi\ o \to \Phi\ o \to \Phi\ o$.
PARAMETER *impv* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_{\mathcal{O}}\ o \to \checkmark_{\Phi}\ \phi_1 \to \checkmark_{\Phi}\ \phi_2 \to \checkmark_{\Phi}\ (\supset\ \phi_1\ \phi_2)$.
PARAMETER *impi* : $\forall\ o : \mathcal{O},\ \forall\ \phi\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_{\mathcal{O}}\ o \to \checkmark_{\Phi}\ \phi \to \checkmark_{\Phi}\ \phi_1 \to \checkmark_{\Phi}\ \phi_2 \to \vdash\ (\wedge\ \phi\ \phi_1)\ \phi_2 \to \vdash\ \phi\ (\supset\ \phi_1\ \phi_2)$.
PARAMETER *impe* : $\forall\ o : \mathcal{O},\ \forall\ \phi_1\ \phi_2 : \Phi\ o,\ \checkmark_{\mathcal{O}}\ o \to \checkmark_{\Phi}\ \phi_1 \to \checkmark_{\Phi}\ \phi_2 \to \vdash\ (\wedge\ (\supset\ \phi_1\ \phi_2)\ \phi_1)\ \phi_2$.
PARAMETER *imps* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_2,\ \forall\ \phi_1\ \phi_2 : \Phi\ o_2,\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \swarrow_{\rightsquigarrow}\ m \to \checkmark_{\Phi}\ \phi_1 \to \checkmark_{\Phi}\ \phi_2 \to \vdash\ (\supset\ (\dot{\phi}[\cdot]\ m\ \phi_1)\ (\dot{\phi}[\cdot]\ m\ \phi_2))\ (\dot{\phi}[\cdot]\ m\ (\supset\ \phi_1\ \phi_2))$.

Universal Quantification

PARAMETER $\forall : \forall\ \{o_1 : \mathcal{O}\},\ \forall\ o_2 : \mathcal{O},\ \Phi\ (\times\ o_1\ o_2) \to \Phi\ o_1$.
PARAMETER *sforallv* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ \phi_{12} : \Phi\ (\times\ o_1\ o_2),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \checkmark_{\Phi}\ \phi_{12} \to \checkmark_{\Phi}\ (\forall\ o_2\ \phi_{12})$.
PARAMETER *sforalli* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ \phi_1 : \Phi\ o_1,\ \forall\ \phi_{12} : \Phi\ (\times\ o_1\ o_2),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \checkmark_{\Phi}\ \phi_1 \to \checkmark_{\Phi}\ \phi_{12} \to \vdash\ (\dot{\phi}[\cdot]\ (\pi_1\ o_1\ o_2)\ \phi_1)\ \phi_{12} \to \vdash\ \phi_1\ (\forall\ o_2\ \phi_{12})$.
PARAMETER *sforalle* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ \phi_1 : \Phi\ o_1,\ \forall\ \phi_{12} : \Phi\ (\times\ o_1\ o_2),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \checkmark_{\Phi}\ \phi_1 \to \checkmark_{\Phi}\ \phi_{12} \to \vdash\ \phi_1\ (\forall\ o_2\ \phi_{12}) \to \vdash\ (\dot{\phi}[\cdot]\ (\pi_1\ o_1\ o_2)\ \phi_1)\ \phi_{12}$.
PARAMETER *sforalls* : $\forall\ o_1\ o_1'\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_1',\ \forall\ \phi_{12}' : \Phi\ (\times\ o_1'\ o_2),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_1' \to \checkmark_{\mathcal{O}}\ o_2 \to \swarrow_{\rightsquigarrow}\ m \to \checkmark_{\Phi}\ \phi_{12}' \to \vdash\ (\forall\ o_2\ (\dot{\phi}[\cdot]\ ((\langle\cdot,\cdot\rangle\ (;\ (\pi_1\ o_1\ o_2)\ m)\ (\pi_2\ o_1\ o_2))\ \phi_{12}'))\ (\dot{\phi}[\cdot]\ m\ (\forall\ o_2\ \phi_{12}'))$.

Existential Quantification

PARAMETER $\exists : \forall\ \{o_1 : \mathcal{O}\},\ \forall\ o_2 : \mathcal{O},\ \Phi\ (\times\ o_1\ o_2) \to \Phi\ o_1$.
PARAMETER *sexistsv* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ \phi_{12} : \Phi\ (\times\ o_1\ o_2),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \checkmark_{\Phi}\ \phi_{12} \to \checkmark_{\Phi}\ (\exists\ o_2\ \phi_{12})$.
PARAMETER *sexistsi* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ \phi_{12} : \Phi\ (\times\ o_1\ o_2),\ \forall\ \phi_1 : \Phi\ o_1,\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \checkmark_{\Phi}\ \phi_{12} \to \checkmark_{\Phi}\ \phi_1 \to \vdash\ \phi_{12}\ (\dot{\phi}[\cdot]\ (\pi_1\ o_1\ o_2)\ \phi_1) \to \vdash\ (\exists\ o_2\ \phi_{12})\ \phi_1$.
PARAMETER *sexistse* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ \phi_{12} : \Phi\ (\times\ o_1\ o_2),\ \forall\ \phi_1 : \Phi\ o_1,\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \checkmark_{\Phi}\ \phi_{12} \to \checkmark_{\Phi}\ \phi_1 \to \vdash\ (\exists\ o_2\ \phi_{12})\ \phi_1 \to \vdash\ \phi_{12}\ (\dot{\phi}[\cdot]\ (\pi_1\ o_1\ o_2)\ \phi_1)$.
PARAMETER *sexistss* : $\forall\ o_1\ o_1'\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_1',\ \forall\ \phi_{12}' : \Phi\ (\times\ o_1'\ o_2),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_1' \to \checkmark_{\mathcal{O}}\ o_2 \to \swarrow_{\rightsquigarrow}\ m \to \checkmark_{\Phi}\ \phi_{12}' \to \vdash\ (\dot{\phi}[\cdot]\ m\ (\exists\ o_2\ \phi_{12}'))\ (\exists\ o_2\ (\dot{\phi}[\cdot]\ ((\cdot,\cdot\rangle\ (;\ (\pi_1\ o_1\ o_2)\ m)\ (\pi_2\ o_1\ o_2))\ \phi_{12}'))$.

Equality

PARAMETER $= : \forall\ \{o_1 : \mathcal{O}\},\ \forall\ o_2 : \mathcal{O},\ \Phi\ (\times\ o_1\ o_2) \to \Phi\ (\times\ o_1\ (\times\ o_2\ o_2))$.
PARAMETER *seqv* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ \phi_{12} : \Phi\ (\times\ o_1\ o_2),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \checkmark_{\Phi}\ \phi_{12} \to \checkmark_{\Phi}\ (=\ o_2\ \phi_{12})$.
PARAMETER *seqi* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ \phi_{12} : \Phi\ (\times\ o_1\ o_2),\ \forall\ \phi_{122} : \Phi\ (\times\ o_1\ (\times\ o_2\ o_2)),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \checkmark_{\Phi}\ \phi_{12} \to \checkmark_{\Phi}\ \phi_{122} \to \vdash\ \phi_{12}\ (\dot{\phi}[\cdot]\ ((\langle\cdot,\cdot\rangle\ (\pi_1\ o_1\ o_2)\ (\langle\cdot,\cdot\rangle\ (\pi_2\ o_1\ o_2)\ (\pi_2\ o_1\ o_2)))\ \phi_{122})) \to \vdash\ (=\ o_2\ \phi_{12})\ \phi_{122}$.
PARAMETER *seqe* : $\forall\ o_1\ o_2 : \mathcal{O},\ \forall\ \phi_{12} : \Phi\ (\times\ o_1\ o_2),\ \forall\ \phi_{122} : \Phi\ (\times\ o_1\ (\times\ o_2\ o_2)),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_2 \to \checkmark_{\Phi}\ \phi_{12} \to \checkmark_{\Phi}\ \phi_{122} \to \vdash\ (=\ o_2\ \phi_{12})\ \phi_{122} \to \vdash\ \phi_{12}\ (\dot{\phi}[\cdot]\ ((\langle\cdot,\cdot\rangle\ (\pi_1\ o_1\ o_2)\ ((\langle\cdot,\cdot\rangle\ (\pi_2\ o_1\ o_2)\ (\pi_2\ o_1\ o_2)))\ \phi_{122}))$.
PARAMETER *seqs* : $\forall\ o_1\ o_1'\ o_2 : \mathcal{O},\ \forall\ m : \rightsquigarrow\ o_1\ o_1',\ \forall\ \phi_{12}' : \Phi\ (\times\ o_1'\ o_2),\ \checkmark_{\mathcal{O}}\ o_1 \to \checkmark_{\mathcal{O}}\ o_1' \to \checkmark_{\mathcal{O}}\ o_2 \to \swarrow_{\rightsquigarrow}\ m \to \checkmark_{\Phi}\ \phi_{12}' \to \vdash\ (\dot{\phi}[\cdot]\ ((\langle\cdot,\cdot\rangle\ (;\ (\pi_1\ o_1\ (\times\ o_2\ o_2))\ m)\ (\pi_2\ o_1\ (\times\ o_2\ o_2)))\ (=\ o_2\ \phi_{12}'))\ (=\ o_2\ (\dot{\phi}[\cdot]\ ((\langle\cdot,\cdot\rangle\ (;\ (\pi_1\ o_1\ o_2)\ m)\ (\pi_2\ o_1\ o_2))\ \phi_{12}'))$.

Impredicativity

PARAMETER $\Omega : \mathcal{O}$.
PARAMETER *holds* : $\Phi\ \Omega$.
PARAMETER $\chi : \forall\ \{o : \mathcal{O}\},\ \Phi\ o \to \rightsquigarrow\ o\ \Omega$.
PARAMETER *opropv* : $\checkmark_{\mathcal{O}}\ \Omega$.
PARAMETER *holdsv* : $\checkmark_{\Phi}\ holds$.
PARAMETER *characterv* : $\forall\ o : \mathcal{O},\ \forall\ \phi : \Phi\ o,\ \checkmark_{\mathcal{O}}\ o \to \checkmark_{\Phi}\ \phi \to \swarrow_{\rightsquigarrow}\ (\chi\ \phi)$.
PARAMETER *holdsi* : $\forall\ o : \mathcal{O},\ \forall\ \phi : \Phi\ o,\ \checkmark_{\mathcal{O}}\ o \to \checkmark_{\Phi}\ \phi \to \vdash\ \phi\ (\dot{\phi}[\cdot]\ (\chi\ \phi)\ holds)$.
PARAMETER *holdse* : $\forall\ o : \mathcal{O},\ \forall\ \phi : \Phi\ o,\ \checkmark_{\mathcal{O}}\ o \to \checkmark_{\Phi}\ \phi \to \vdash\ (\dot{\phi}[\cdot]\ (\chi\ \phi)\ holds)\ \phi$.

END *HigherOrderFibration*.

When a higher-order fibration is defined over a category with a natural-number object $\boldsymbol{N}$, it automatically has an $\mathbb{m}$ predicate over $\boldsymbol{N}$. However, the automatic construction of $\mathbb{m}$ tends to be complicated, so here we allow one to specify an optimized, necessarily equivalent, construction.

MODULE TYPE *NatHigherOrderFibration* (*CCC* : *CartesianClosedCategory*) (*NNO* : *NaturalNumberObject CCC*).
INCLUDE *HigherOrderFibration CCC*.

PARAMETER $\mathbb{m} : \Phi\ \boldsymbol{N}$.
PARAMETER *isnatv* : $\checkmark_{\Phi}\ \mathbb{m}$.
PARAMETER *isnatiz* : $\vdash\ (\top\ \mathbf{1})\ (\dot{\phi}[\cdot]\ \boldsymbol{Z}\ \mathbb{m})$.
PARAMETER *isnatis* : $\vdash\ \mathbb{m}\ (\dot{\phi}[\cdot]\ \boldsymbol{S}\ \mathbb{m})$.
PARAMETER *isnate* : $\forall\ \phi : \Phi\ \boldsymbol{N},\ \checkmark_{\Phi}\ \phi \to \vdash\ (\top\ \mathbf{1})\ (\dot{\phi}[\cdot]\ \boldsymbol{Z}\ \phi) \to \vdash\ \phi\ (\dot{\phi}[\cdot]\ \boldsymbol{S}\ \phi) \to \vdash\ \mathbb{m}\ \phi$.

END *NatHigherOrderFibration*.

D-B. RCAstoHOFs.v

D-B1. Proof that a Relational Combinatory Algebra forms a Higher-Order Fibration

This module demonstrates that every relational (and partial) combinatory algebra forms a consistent higher-order fibration over the inhabited sets, as claimed in Section 4.2 (and Section 3.2). We only show the definitions of the constructions and the realizers of the various entailments and omit the lemmas required by *NatHigherOrderFibration*, the proofs of which are straightforward from the definitions.

MODULE *RCAtoHOF* (*RCA* : *RelationalCombinatoryAlgebra*) <: *NatHigherOrderFibration InhabitedSets InhabitedSetsNat*.

DEFINITION $\Phi\ (o : \mathcal{O}) : \text{TYPE} := set\ o \to C \to \text{PROP}$.
DEFINITION $\checkmark_{\Phi}\ \{o : \mathcal{O}\}\ (\phi : \Phi\ o) : \text{PROP} := \forall\ x\ x' : set\ o,\ \forall\ c : C,\ \checkmark_{set}\ o\ x \to \checkmark_{set}\ o\ x' \to \approx_{set}\ o\ x\ x' \to \checkmark_{C}\ c \to \phi\ x\ c \to \phi\ x'\ c$.

DEFINITION $\vdash\ \{o : \mathcal{O}\}\ (\phi_1\ \phi_2 : \Phi\ o) : \text{PROP} := \exists\ c : C,\ \checkmark_{C}\ c \wedge \forall\ x : set\ o,\ \forall\ c1 : C,\ \checkmark_{set}\ o\ x \to \checkmark_{C}\ c1 \to \phi_1\ x\ c1 \to \downarrow_{\phi}\ \downarrow_{c}\ c\ \downarrow\ c\ c1\ (\phi_2\ x)$.

DEFINITION $\subseteq_{\Phi}\ \{o : \mathcal{O}\}\ (\phi_1\ \phi_2 : \Phi\ o) : \text{PROP} := \forall\ x : set\ o,\ \forall\ c : C,\ \checkmark_{set}\ o\ x \to \checkmark_{C}\ c \to \phi_1\ x\ c \to \phi_2\ x\ c$.
DEFINITION *caxiom* : $C := c_{\lambda}\ 0\ (evar\ 1\ None)$.
DEFINITION *ccut* (*c1 c2* : *C*) : $C := c_{\lambda}\ 0\ (\cdot\ ((\cdot)\ c2)\ (\cdot\ ((\cdot)\ c1)\ (evar\ 1\ None)))$.

DEFINITION $\dot{\phi}[\cdot]\ \{o_1\ o_2 : \mathcal{O}\}\ (m : \rightsquigarrow\ o_1\ o_2)\ (\phi_2 : \Phi\ o_2) : \Phi\ o_1 := \lambda\ s1 \mapsto \phi_2\ (m\ s1)$.

DEFINITION $\top\ (o : \mathcal{O}) : \Phi\ o := \lambda\ x\ c \mapsto \text{TRUE}$.

Definition *c2sel1* : $C := c_\lambda\ 1\ (evar\ 2\ None)$.
Definition *c2sel2* : $C := c_\lambda\ 1\ (evar\ 2\ (Some\ None))$.
Definition $\wedge\ \{o : \mathcal{O}\}\ (\phi_1\ \phi_2 : \Phi\ o) : \Phi\ o := \lambda\, x\ c \mapsto \downarrow_\phi \downarrow_c \downarrow c\ c2sel1\ (\phi_1\ x) \wedge \downarrow_\phi \downarrow_c \downarrow c\ c2sel2\ (\phi_2\ x)$.
Definition *cconji* $(e1\ e2 : C) : C := c_\lambda\ 1\ (\cdot\ (\cdot\ (evar\ 2\ (Some\ None))\ (\cdot\ ((\cdot)\ e1)\ (evar\ 2\ None)))\ (\cdot\ ((\cdot)\ e2)\ (evar\ 2\ None)))$.
Definition *cconje1* : $C := c_\lambda\ 0\ (\cdot\ (evar\ 1\ None)\ ((\cdot)\ c2sel1))$.
Definition *cconje2* : $C := c_\lambda\ 0\ (\cdot\ (evar\ 1\ None)\ ((\cdot)\ c2sel2))$.

Definition $\bot\ (o : \mathcal{O}) : \Phi\ o := \lambda\, x\ c \mapsto$ False.

Definition $\vee\ \{o : \mathcal{O}\}\ (\phi_1\ \phi_2 : \Phi\ o) : \Phi\ o := \lambda\, x\ c \mapsto (\exists\ c1 : C, \not\downarrow_C\ c1 \wedge \phi_1\ x\ c1 \wedge c_\lambda\ 1\ (\cdot\ (evar\ 2\ None)\ ((\cdot)\ c1)) = c) \vee (\exists\ c2 : C, \not\downarrow_C\ c2 \wedge \phi_2\ x\ c2 \wedge c_\lambda\ 1\ (\cdot\ (evar\ 2\ (Some\ None))\ ((\cdot)\ c2)) = c)$.
Definition *cdisji1* : $C := c_\lambda\ 2\ (\cdot\ (evar\ 3\ (Some\ None))\ (evar\ 3\ None))$.
Definition *cdisji2* : $C := c_\lambda\ 2\ (\cdot\ (evar\ 3\ (Some\ (Some\ None)))\ (evar\ 3\ None))$.
Definition *cdisje* $(e1\ e2 : C) : C := c_\lambda\ 0\ (\cdot\ (\cdot\ (evar\ 1\ None)\ ((\cdot)\ e1))\ ((\cdot)\ e2))$.

Definition $\supset\ \{o : \mathcal{O}\}\ (\phi_1\ \phi_2 : \Phi\ o) : \Phi\ o := \lambda\, x\ c \mapsto \forall\ c1 : C, \not\downarrow_C\ c1 \to \phi_1\ x\ c1 \to \downarrow_\phi \downarrow_c \downarrow c\ c1\ (\phi_2\ x)$.
Definition *ctuple2* : $C := c_\lambda\ 2\ (\cdot\ (\cdot\ (evar\ 3\ (Some\ (Some\ None)))\ (evar\ 3\ None))\ (evar\ 3\ (Some\ None)))$.
Definition *cimpi* $(e : C) : C := c_\lambda\ 1\ (\cdot\ ((\cdot)\ e)\ (\cdot\ (\cdot\ ((\cdot)\ ctuple2)\ (evar\ 2\ None))\ (evar\ 2\ (Some\ None))))$.
Definition *cimpe* : $C := c_\lambda\ 0\ (\cdot\ (\cdot\ (evar\ 1\ None)\ ((\cdot)\ c2sel1))\ (\cdot\ (evar\ 1\ None)\ ((\cdot)\ c2sel2)))$.

Definition $\forall\ \{o_1 : \mathcal{O}\}\ (o_2 : \mathcal{O})\ (\phi_{12} : \Phi\ (\times\ o_1\ o_2)) : \Phi\ o_1 := \lambda\, x1\ c \mapsto \forall\ x2 : set\ o_2, \not\downarrow_{set}\ o_2\ x2 \to \phi_{12}\ (pair\ x1\ x2)\ c$.

Definition $\exists\ \{o_1 : \mathcal{O}\}\ (o_2 : \mathcal{O})\ (\phi_{12} : \Phi\ (\times\ o_1\ o_2)) : \Phi\ o_1 := \lambda\, x1\ c \mapsto \exists\ x2 : set\ o_2, \not\downarrow_{set}\ o_2\ x2 \wedge \phi_{12}\ (pair\ x1\ x2)\ c$.

Definition $=\ \{o_1 : \mathcal{O}\}\ (o_2 : \mathcal{O})\ (\phi_{12} : \Phi\ (\times\ o_1\ o_2)) : \Phi\ (\times\ o_1\ (\times\ o_2\ o_2))$
$:= \lambda\, x122\ c \mapsto \approx_{set}\ o_2\ (fst\ (snd\ x122))\ (snd\ (snd\ x122)) \wedge \phi_{12}\ (pair\ (fst\ x122)\ (fst\ (snd\ x122)))\ c$.

Definition $\Omega : \mathcal{O} :=$ {| *set* $:= C \to$ Prop; $\not\downarrow_{set} := \lambda\, \phi \mapsto \forall\ c : C, \not\downarrow_C\ c \to \phi\ c \to \phi\ c$; $\approx_{set} := \lambda\, \phi\ p' \mapsto \forall\ c : C, \not\downarrow_C\ c \to \phi\ c \leftrightarrow p'\ c$ |}.
Definition *holds* : $\Phi\ \Omega := \lambda\, \phi \mapsto \phi$.
Definition $\chi\ \{o : \mathcal{O}\}\ (\phi : \Phi\ o) : \rightsquigarrow\ o\ \Omega := \phi$.

Fixpoint $c_\cdot^\lambda\ (n : \mathbb{N}) : C$

$:=$ match n with $0 \mapsto c_\lambda\ 1\ (evar\ 2\ (Some\ None)) \mid S\ n \mapsto c_\lambda\ 1\ (\cdot\ (evar\ 2\ None)\ (\cdot\ (\cdot\ ((\cdot)\ (c_\cdot^\lambda\ n))\ (evar\ 2\ None))\ (evar\ 2\ (Some\ None))))$ end.

Definition $\mathbb{n} : \Phi\ \boldsymbol{N} := \lambda\, n \mapsto eq\ (c_\cdot^\lambda\ n)$.

Definition *cnatiz* : $C := c_\lambda\ 0\ ((\cdot)\ (c_\cdot^\lambda\ 0))$.
Definition *cnatis* : $C := c_\lambda\ 2\ (\cdot\ (evar\ 3\ (Some\ None))\ (\cdot\ (\cdot\ (evar\ 3\ None)\ (evar\ 3\ (Some\ None)))\ (evar\ 3\ (Some\ (Some\ None)))))$.
Definition *cnate* $(cz\ cs : C) : C := c_\lambda\ 0\ (\cdot\ (\cdot\ ((\cdot)\ (c_\lambda\ 1\ (\cdot\ (\cdot\ (evar\ 2\ None)\ ((\cdot)\ cs))\ (evar\ 2\ (Some\ None)))))\ (evar\ 1\ None))\ (\cdot\ ((\cdot)\ cz)\ ((\cdot)\ caxiom)))$.

Inductive *Realizable* $(\phi : \Phi\ \mathbf{1}) :$ Prop := *realizable* $(c : C) : \not\downarrow_C\ c \to \phi\ tt\ c \to Realizable\ \phi$.

Theorem *entails_realizable* $(\phi_1\ \phi_2 : \Phi\ \mathbf{1}) : \not\Phi\ \phi_1 \to \not\Phi\ \phi_2 \to \vdash \phi_1\ \phi_2 \to Realizable\ \phi_1 \to Realizable\ \phi_2$.

Theorem *consistent* : $\vdash (\top\ \mathbf{1})\ (\bot\ \mathbf{1}) \to$ False.

End *RCAtoHOF*.

D-C. SCAstoHOFs.v

D-C1. Proof that a Stateful Combinatory Algebra forms a Higher-Order Fibration

This module demonstrates that every stateful combinatory algebra forms a consistent higher-order fibration over the inhabited sets, as claimed in Theorem 5.3. It is the formal statement of Figure 2. We only show the definitions of the constructions and omit the lemmas required by *NatHigherOrderFibration*, the proofs of which are straightforward from the definitions (and the realizers of entailment for which are the same as for RCAs).

Module *SCAtoHOF* (*SCA* : *StatefulCombinatoryAlgebra*) <: *NatHigherOrderFibration InhabitedSets InhabitedSetsNat*.

Definition $\Phi\ (o : \mathcal{O}) :$ Type $:=$ *set* $o \to \Sigma \to C \to$ Prop.
Record $\not\Phi\ \{o : \mathcal{O}\}\ (\hat{\phi} : \Phi\ o) :$ Prop
$:=$
{ *propv* : $\forall\ x : set\ o, \forall\ s\ s' : \Sigma, \forall\ c : C, \not\downarrow_{set}\ o\ x \to \not\downarrow_\Sigma\ s \to \not\downarrow_\Sigma\ s' \to \leq\ s\ s' \to \not\downarrow_C\ s\ c \to \hat{\phi}\ x\ s\ c \to \hat{\phi}\ x\ s'\ c$
; *prope* : $\forall\ x\ x' : set\ o, \forall\ s : \Sigma, \forall\ c : C, \not\downarrow_{set}\ o\ x \to \not\downarrow_{set}\ o\ x' \to \approx_{set}\ o\ x\ x' \to \not\downarrow_\Sigma\ s \to \not\downarrow_C\ s\ c \to \hat{\phi}\ x\ s\ c \to \hat{\phi}\ x'\ s\ c$
}.

Definition $\vdash\ \{o : \mathcal{O}\}\ (\hat{\phi}_1\ \hat{\phi}_2 : \Phi\ o) :$ Prop $:= \exists\ c : C, \forall\ s : \Sigma, \not\downarrow_\Sigma\ s \to \not\downarrow_C\ s\ c \wedge \forall\ x : set\ o, \forall\ s : \Sigma, \forall\ c1 : C, \not\downarrow_{set}\ o\ x \to \not\downarrow_\Sigma\ s \to \not\downarrow_C\ s\ c1 \to \hat{\phi}_1\ x\ s\ c1 \to \downarrow_\phi \downarrow_c \downarrow s\ c\ c1\ (\hat{\phi}_2\ x)$.

Definition $\subseteq_\Phi\ \{o : \mathcal{O}\}\ (\hat{\phi}_1\ \hat{\phi}_2 : \Phi\ o) :$ Prop $:= \forall\ x : set\ o, \forall\ s : \Sigma, \forall\ c : C, \not\downarrow_{set}\ o\ x \to \not\downarrow_\Sigma\ s \to \not\downarrow_C\ s\ c \to \hat{\phi}_1\ x\ s\ c \to \hat{\phi}_2\ x\ s\ c$.

Definition $\dot{\phi}[\cdot]\ \{o_1\ o_2 : \mathcal{O}\}\ (m : \rightsquigarrow\ o_1\ o_2)\ (\hat{\phi}_2 : \Phi\ o_2) : \Phi\ o_1 := \lambda\, s1 \mapsto \hat{\phi}_2\ (m\ s1)$.

Definition $\top\ (o : \mathcal{O}) : \Phi\ o := \lambda\, x\ s\ c \mapsto$ True.

Definition *c2sel1* : $C := c_\lambda\ 1\ (evar\ 2\ None)$.
Definition *c2sel2* : $C := c_\lambda\ 1\ (evar\ 2\ (Some\ None))$.
Definition $\wedge\ \{o : \mathcal{O}\}\ (\hat{\phi}_1\ \hat{\phi}_2 : \Phi\ o) : \Phi\ o := \lambda\, x\ s\ c \mapsto \forall\ s' : \Sigma, \not\downarrow_\Sigma\ s' \to \leq\ s\ s' \to \downarrow_\phi \downarrow_c \downarrow s'\ c\ c2sel1\ (\hat{\phi}_1\ x) \wedge \downarrow_\phi \downarrow_c \downarrow s'\ c\ c2sel2\ (\hat{\phi}_2\ x)$.

Definition $\bot\ (o : \mathcal{O}) : \Phi\ o := \lambda\, x\ s\ c \mapsto$ False.

Definition $\vee\ \{o : \mathcal{O}\}\ (\hat{\phi}_1\ \hat{\phi}_2 : \Phi\ o) : \Phi\ o := \lambda\, x\ s\ c \mapsto (\exists\ c1 : C, \not\downarrow_C\ s\ c1 \wedge \hat{\phi}_1\ x\ s\ c1 \wedge c_\lambda\ 1\ (\cdot\ (evar\ 2\ None)\ ((\cdot)\ c1)) = c) \vee (\exists\ c2 : C, \not\downarrow_C\ s\ c2 \wedge \hat{\phi}_2\ x\ s\ c2 \wedge c_\lambda\ 1\ (\cdot\ (evar\ 2\ (Some\ None))\ ((\cdot)\ c2)) = c)$.

Definition $\supset\ \{o : \mathcal{O}\}\ (\hat{\phi}_1\ \hat{\phi}_2 : \Phi\ o) : \Phi\ o := \lambda\, x\ s\ c \mapsto \forall\ s' : \Sigma, \forall\ c1 : C, \not\downarrow_\Sigma\ s' \to \leq\ s\ s' \to \not\downarrow_C\ s'\ c1 \to \hat{\phi}_1\ x\ s'\ c1 \to \downarrow_\phi \downarrow_c \downarrow s'\ c\ c1\ (\hat{\phi}_2\ x)$.

Definition $\forall\ \{o_1 : \mathcal{O}\}\ (o_2 : \mathcal{O})\ (\hat{\phi}_{12} : \Phi\ (\times\ o_1\ o_2)) : \Phi\ o_1 := \lambda\, x1\ s\ c \mapsto \forall\ x2 : set\ o_2, \not\downarrow_{set}\ o_2\ x2 \to \hat{\phi}_{12}\ (pair\ x1\ x2)\ s\ c$.

Definition $\exists\ \{o_1 : \mathcal{O}\}\ (o_2 : \mathcal{O})\ (\hat{\phi}_{12} : \Phi\ (\times\ o_1\ o_2)) : \Phi\ o_1 := \lambda\, x1\ s\ c \mapsto \exists\ x2 : set\ o_2, \not\downarrow_{set}\ o_2\ x2 \wedge \hat{\phi}_{12}\ (pair\ x1\ x2)\ s\ c$.

Definition $=\ \{o_1 : \mathcal{O}\}\ (o_2 : \mathcal{O})\ (\hat{\phi}_{12} : \Phi\ (\times\ o_1\ o_2)) : \Phi\ (\times\ o_1\ (\times\ o_2\ o_2))$
$:= \lambda\, x122\ s\ c \mapsto \approx_{set}\ o_2\ (fst\ (snd\ x122))\ (snd\ (snd\ x122)) \wedge \hat{\phi}_{12}\ (pair\ (fst\ x122)\ (fst\ (snd\ x122)))\ s\ c$.

Definition $\Omega : \mathcal{O}$
$:=$ {| *set* $:= \Sigma \to C \to$ Prop
; $\not\downarrow_{set} := \lambda\, \hat{\phi} \mapsto \forall\ s\ s' : \Sigma, \forall\ c : C, \not\downarrow_\Sigma\ s \to \not\downarrow_\Sigma\ s' \to \leq\ s\ s' \to \not\downarrow_C\ s\ c \to \hat{\phi}\ s\ c \to \hat{\phi}\ s'\ c$

; $\approx_{set} := \lambda\, \hat{\phi}\ p' \mapsto \forall\ s : \Sigma,\ \forall\ c : C,\ \checkmark_{\Sigma}\ s \to \checkmark_{C}\ s\ c \to \hat{\phi}\ s\ c \leftrightarrow p'\ s\ c\ |\}$.

Definition *holds* : $\Phi\ \Omega := \lambda\, \hat{\phi} \mapsto \hat{\phi}$.

Definition χ $\{o : \mathcal{O}\}$ $(\hat{\phi} : \Phi\ o) : \rightsquigarrow\ o\ \Omega := \hat{\phi}$.

Fixpoint $c^{\lambda}_{\cdot}$ $(n : \mathbb{N}) : C$

:= match n with $0 \mapsto c_{\lambda}\ 1\ (evar\ 2\ (Some\ None)) \mid S\ n \mapsto c_{\lambda}\ 1\ (\cdot\ (evar\ 2\ None)\ (\cdot\ (\cdot\ ((\cdot)\ (c^{\lambda}_{\cdot}\ n))\ (evar\ 2\ None))\ (evar\ 2\ (Some\ None))))$ end.

Definition $\mathbb{m} : \Phi\ \boldsymbol{N} := \lambda\, n\ \_ \mapsto eq\ (c^{\lambda}_{\cdot}\ n)$.

Inductive *Realizable* $(\hat{\phi} : \Phi\ \mathbf{1}) : \mathrm{Prop} := realizable\ (s : \Sigma)\ (c : C) : \checkmark_{\Sigma}\ s \to \checkmark_{C}\ s\ c \to \hat{\phi}\ tt\ s\ c \to Realizable\ \hat{\phi}$.

Theorem *entails_realizable* $(\hat{\phi}_1\ \hat{\phi}_2 : \Phi\ \mathbf{1}) : \checkmark_{\Phi}\ \hat{\phi}_1 \to \checkmark_{\Phi}\ \hat{\phi}_2 \to \vdash \hat{\phi}_1\ \hat{\phi}_2 \to Realizable\ \hat{\phi}_1 \to Realizable\ \hat{\phi}_2$.

Theorem *consistent* : $\vdash (\top\ \mathbf{1})\ (\bot\ \mathbf{1}) \to \mathrm{False}$.

End *SCAtoHOF*.

E Countable Choice

E-A. CountableChoice.v

E-A1. Definition of Internal Countable Choice in Evidenced Frames

This is the formal statement of Definition 2.2. It is in point-free notation because it is defined over any cartesian-closed category, not just sets.

Module *CCinHOF* (*CCC* : *CartesianClosedCategory*) (*NNO* : *NaturalNumberObject CCC*) (*HOF* : *NatHigherOrderFibration CCC NNO*).

Definition *total* $(o_1\ o_2 : \mathcal{O})\ (\phi_1 : \Phi\ o_1) : \Phi\ (\Rightarrow o_1\ (\Rightarrow o_2\ \Omega))$

:= $\forall\ o_1\ (\supset\ (\dot{\phi}[\cdot]\ \pi_2\ \phi_1)\ (\dot{\phi}[\cdot]\ (\boldsymbol{eval}\ o_1\ (\Rightarrow o_2\ \Omega))\ (\exists\ o_2\ (\dot{\phi}[\cdot]\ (\boldsymbol{eval}\ o_2\ \Omega)\ holds))))$.

Definition *includes* : $\Phi\ (\times\ \Omega\ \Omega) := \supset\ (\dot{\phi}[\cdot]\ \pi_2\ holds)\ (\dot{\phi}[\cdot]\ \pi_1\ holds)$.

Definition *lift* $(o_1\ o_2 : \mathcal{O})\ (\phi_2 : \Phi\ (\times\ o_2\ o_2)) : \Phi\ (\times\ (\Rightarrow o_1\ o_2)\ (\Rightarrow o_1\ o_2))$

:= $\forall\ o_1\ (\dot{\phi}[\cdot]\ (\langle\cdot,\cdot\rangle\ (;\ (\langle\cdot,\cdot\rangle\ (;\ \pi_1\ \pi_1)\ \pi_2)\ (\boldsymbol{eval}\ o_1\ o_2))\ (;\ (\langle\cdot,\cdot\rangle\ (;\ \pi_1\ \pi_2)\ \pi_2)\ (\boldsymbol{eval}\ o_1\ o_2)))\ \phi_2)$.

Definition *determined* $(o_1\ o_2 : \mathcal{O}) : \Phi\ (\Rightarrow o_1\ (\Rightarrow o_2\ \Omega))$

:= $\forall\ o_1\ (\dot{\phi}[\cdot]\ (\boldsymbol{eval}\ o_1\ (\Rightarrow o_2\ \Omega))\ (\forall\ o_2\ (\forall\ o_2\ (\supset\ (\dot{\phi}[\cdot]\ (;\ (\langle\cdot,\cdot\rangle\ (;\ \pi_1\ \pi_1)\ (;\ \pi_1\ \pi_2))\ (\boldsymbol{eval}\ o_2\ \Omega))\ holds)$
$(\supset\ (\dot{\phi}[\cdot]\ (;\ (\langle\cdot,\cdot\rangle\ (;\ \pi_1\ \pi_1)\ \pi_2)\ (\boldsymbol{eval}\ o_2\ \Omega))\ holds)$
$(\dot{\phi}[\cdot]\ (\langle\cdot,\cdot\rangle\ !\ (\langle\cdot,\cdot\rangle\ (;\ \pi_1\ \pi_2)\ \pi_2))\ (=\ o_2\ (\top\ (\times\ \mathbf{1}\ o_2))))))))))$.

Definition *countable_choice* $(o : \mathcal{O}) : \Phi\ \mathbf{1}$

:= $\forall\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))\ (\dot{\phi}[\cdot]\ \pi_2\ (\supset\ (total\ \boldsymbol{N}\ o\ \mathbb{m})$
$(\exists\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))\ (\wedge\ (\wedge\ (\dot{\phi}[\cdot]\ \pi_2\ (total\ \boldsymbol{N}\ o\ \mathbb{m}))\ (lift\ \boldsymbol{N}\ (\Rightarrow o\ \Omega)\ (lift\ o\ \Omega\ includes)))$
$(\dot{\phi}[\cdot]\ \pi_2\ (determined\ \boldsymbol{N}\ o)))))))$.

End *CCinHOF*.

E-A2. Definition of Internal Countable Choice in Relational Combinatory Algebras

This is the formal statement of the definition in Lemmas 3.3 and 4.5.

Module *CCinRCA* (*RCA* : *RelationalCombinatoryAlgebra*).

Definition $\exists|\ \{o_1 : \mathcal{O}\}\ (o_2 : \mathcal{O})\ (P12 : set\ o_1 \to set\ o_2 \to \mathrm{Prop})\ (p12 : \Phi\ (\times\ o_1\ o_2)) : \Phi\ o_1$
:= $\lambda\, x1\ c \mapsto \exists\ x2 : set\ o_2,\ \checkmark_{set}\ o_2\ x2 \wedge P12\ x1\ x2 \wedge p12\ (pair\ x1\ x2)\ c$.

Definition *rca_included* $\{o : \mathcal{O}\}\ (S\ R : set\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))) : \mathrm{Prop} := \forall\ n : \mathbb{N},\ \forall\ x : set\ o,\ \forall\ c : C,\ \checkmark_{set}\ o\ x \to \checkmark_{C}\ c \to S\ n\ x\ c \to R\ n\ x\ c$.

Definition *rca_determined* $\{o : \mathcal{O}\}\ (S : set\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))) : \mathrm{Prop}$
:= $\forall\ n : \mathbb{N},\ \forall\ x\ x' : set\ o,\ \forall\ c\ c' : C,\ \checkmark_{set}\ o\ x \to \checkmark_{set}\ o\ x' \to \checkmark_{C}\ c \to \checkmark_{C}\ c' \to S\ n\ x\ c \to S\ n\ x'\ c' \to \approx_{set}\ o\ x\ x'$.

Definition *rca_countable_choice* $(o : \mathcal{O}) : \Phi\ \mathbf{1}$
:= $\forall\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))\ (\dot{\phi}[\cdot]\ \pi_2\ (\supset\ (total\ \boldsymbol{N}\ o\ \mathbb{m})$
$(\exists|\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))\ (\lambda\, R\ S \mapsto rca\_included\ S\ R \wedge rca\_determined\ S)\ (\dot{\phi}[\cdot]\ \pi_2\ (total\ \boldsymbol{N}\ o\ \mathbb{m})))))$.

E-A3. Proof that Internal CC for a PCA/RCA is Equivalent to Internal CC for its Higher-Order Fibration

We provide the codes exhibiting equivalence, but we elide here the actual proof of Lemmas 3.3 and 4.5 as it is rather tedious and follows from the definitions of the codes.

Definition *csimplify* : $C := c_{\lambda}\ 1\ (\cdot\ ((\cdot)\ (c_{\lambda}\ 1\ (\cdot\ (\cdot\ (evar\ 2\ None)\ ((\cdot)\ c_{snd}))\ (\cdot\ (\cdot\ (evar\ 2\ None)\ ((\cdot)\ c_{fst}))\ (evar\ 2\ (Some\ None))))))\ (\cdot\ (\cdot\ (evar\ 2\ None)\ (evar\ 2\ (Some\ None)))\ ((\cdot)\ c_{fst})))$.

Definition *ccomplicate* : $C := c_{\lambda}\ 1\ (\cdot\ ((\cdot)\ (c_{\lambda}\ 1\ (\cdot\ (\cdot\ (evar\ 2\ (Some\ None))\ (\cdot\ ((\cdot)\ (c_{\lambda}\ 1\ (\cdot\ (\cdot\ (evar\ 2\ (Some\ None))\ (evar\ 2\ None))\ ((\cdot)\ (c_{\lambda}\ 0\ (evar\ 1\ None))))))\ (evar\ 2\ None)))\ ((\cdot)\ (c_{\lambda}\ 2\ (evar\ 3\ (Some\ (Some\ None))))))))\ (\cdot\ (evar\ 2\ None)\ (evar\ 2\ (Some\ None))))))$.

Lemma *rca_countable_choice_equiv* $(o : \mathcal{O}) : \checkmark_{\mathcal{O}}\ o \to \vdash (countable\_choice\ o)\ (rca\_countable\_choice\ o) \wedge \vdash (rca\_countable\_choice\ o)\ (countable\_choice\ o)$.

End *CCinRCA*.

E-A4. Definition of Internal Countable Choice in Stateful Combinatory Algebras

This is the formal statement of the definition in Lemma 5.5.

Module *CCinSCA* (*SCA* : *StatefulCombinatoryAlgebra*).

Definition $\exists|\ \{o_1 : \mathcal{O}\}\ (o_2 : \mathcal{O})\ (P12 : set\ o_1 \to set\ o_2 \to \mathrm{Prop})\ (p12 : \Phi\ (\times\ o_1\ o_2)) : \Phi\ o_1$
:= $\lambda\, x1\ \sigma\ c \mapsto \exists\ x2 : set\ o_2,\ \checkmark_{set}\ o_2\ x2 \wedge P12\ x1\ x2 \wedge p12\ (pair\ x1\ x2)\ \sigma\ c$.

Definition *sca_included* $\{o : \mathcal{O}\}\ (S\ R : set\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))) : \mathrm{Prop}$
:= $\forall\ n : \mathbb{N},\ \forall\ x : set\ o,\ \forall\ \sigma : \Sigma,\ \forall\ c : C,\ \checkmark_{set}\ o\ x \to \checkmark_{\Sigma}\ \sigma \to \checkmark_{C}\ \sigma\ c \to S\ n\ x\ \sigma\ c \to R\ n\ x\ \sigma\ c$.

Definition *sca_determined* $\{o : \mathcal{O}\}\ (S : set\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))) : \mathrm{Prop}$
:= $\forall\ n : \mathbb{N},\ \forall\ x\ x' : set\ o,\ \forall\ \sigma : \Sigma,\ \forall\ c\ c' : C,\ \checkmark_{set}\ o\ x \to \checkmark_{set}\ o\ x' \to \checkmark_{\Sigma}\ \sigma \to \checkmark_{C}\ \sigma\ c \to \checkmark_{C}\ \sigma\ c' \to S\ n\ x\ \sigma\ c \to S\ n\ x'\ \sigma\ c' \to \approx_{set}\ o\ x\ x'$.

Definition *sca_countable_choice* $(o : \mathcal{O}) : \Phi\ \mathbf{1}$
:= $\forall\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))\ (\dot{\phi}[\cdot]\ \pi_2\ (\supset\ (total\ \boldsymbol{N}\ o\ \mathbb{m})$
$(\exists|\ (\Rightarrow \boldsymbol{N}\ (\Rightarrow o\ \Omega))\ (\lambda\, R\ S \mapsto sca\_included\ S\ R \wedge sca\_determined\ S)\ (\dot{\phi}[\cdot]\ \pi_2\ (total\ \boldsymbol{N}\ o\ \mathbb{m})))))$.

E-A5. Proof that Internal CC for an SCA is Equivalent to Internal CC for its Higher-Order Fibration

We provide the codes exhibiting equivalence, but we elide here the actual proof of Lemma 5.5 as it is rather tedious and follows from the definitions of the codes.

```
Definition csimplify : C := c_λ 1 (· ((·) (c_λ 1 (· (· (evar 2 None) ((·) c_snd)) (· (· (evar 2 None) ((·) c_fst)) (evar 2 (Some None))))))
(· (· (evar 2 None) (evar 2 (Some None))) ((·) c_fst))).
Definition ccomplicate : C := c_λ 1 (· ((·) (c_λ 1 (· (· (evar 2 (Some None)) (· ((·) (c_λ 1 (· (· (evar 2 (Some None)) (evar
2 None)) ((·) (c_λ 0 (evar 1 None)))))) (evar 2 None))) ((·) (c_λ 2 (evar 3 (Some (Some None)))))))) (· (evar 2 None) (evar
2 (Some None)))).
Lemma sca_countable_choice_equiv (o : 𝒪) : ✓𝒪 o → ⊢ (countable_choice o) (sca_countable_choice o) ∧ ⊢ (sca_countable_choice
o) (countable_choice o).
End CCinSCA.
```

E-B. CCPCAs.v

E-B1. Proof that Partial Combinatory Algebras Internally Model Countable Choice

This module demonstrates that the higher-order fibration for any PCA internally models countable choice, as claimed in Theorem 3.4. We only show the metatheoretic assumption made and the key lemma enabled by determinism that permits PCAs to model countable choice; we elide here the detailed proof of the theorem.

```
Module CCPCA (PCA : PartialCombinatoryAlgebra).
```

Here we assume the axiom of countable choice in the metatheory.

```
Axiom axiom_of_countable_choice : ∀ I : Set, ∀ ✓ι : I → Prop, ∀ R : ℕ → I → Prop, (∀ n : ℕ, ∃ i : I, ✓ι i ∧ R n i) → ∃
S : ℕ → I → Prop, (∀ n : ℕ, ∀ i : I, S n i → R n i) ∧ (∀ n : ℕ, ∀ i i' : I, S n i → S n i' → i = i') ∧ ∀ n : ℕ, ∃ i : I, ✓ι i
∧ S n i.
```

This is the key lemma that enables us to prove choice. It requires the *red_deterministic* property of PCAs.

```
Lemma red_exists (c_f c_a : C) (I : Set) (✓ι : I → Prop) (ϕ. : I → C → Prop) : ✓𝒞 c_f → ✓𝒞 c_a → (∃ i : I, ✓ι i) → ↓_ϕ ↓_c
↓ c_f c_a (λ c_r ↦ ∃ i : I, ✓ι i ∧ ϕ. i c_r) → ∃ i : I, ✓ι i ∧ ↓_ϕ ↓_c ↓ c_f c_a (ϕ. i).
Definition cc : C := c_λ 0 ((·) caxiom).
Theorem countable_choice (o : 𝒪) : ✓𝒪 o → ⊢ (⊤ 1) (countable_choice o).
End CCPCA.
```

E-C. NCCRCA.v

E-C1. Definition of Flip-RCA

This is the formal definition of Flip-RCA in Figure 1. It uses the *FreeRCA* module for λ-terms, so the following module specifies the primitives and their behavior. Note that primitives here include the $\overline{n}$ codes, with *pvalue* being the corresponding constructor. We use $\mathcal{O}$ here to refer to what the paper refers to as primitives.

```
Module FlipApplicativeStructure <: PrimitiveApplicativeStructure.
  Inductive 𝒪 : Set := const (n : ℕ) | succ | on · do · else· (n : list ℕ) (o_on o_off : 𝒪) | flip.
  Inductive Prim' : Set := pvalue (n : ℕ) | pop (o : 𝒪).
  Definition ✓P (p : P) : Prop := True.
  Inductive ↓𝒪 : 𝒪 → ℕ → ℕ → Prop
  := roconst (n n_a : ℕ) : ↓𝒪 (const n) n_a n
   | rosucc (n : ℕ) : ↓𝒪 succ n (S n)
   | robranchon (n : list ℕ) (o_on o_off : 𝒪) (n_a n_r : ℕ) : In n_a n → ↓𝒪 o_on n_a n_r → ↓𝒪 (on · do · else· n o_on o_off) n_a n_r
   | robranchoff (n : list ℕ) (o_on o_off : 𝒪) (n_a n_r : ℕ) : (In n_a n → False) → ↓𝒪 o_off n_a n_r → ↓𝒪 (on · do · else· n o_on
o_off) n_a n_r
   | rotails (n_a : ℕ) : ↓𝒪 flip n_a 0
   | roheads (n_a : ℕ) : ↓𝒪 flip n_a 1.
  Definition ↓c^p (p_f : P) (c_a : C_? P) (c_r : C_? P) : Prop
  := ∃ of : 𝒪, pop of = p_f ∧ ∃ n_a : ℕ, (·) (pvalue n_a) = c_a ∧ ∃ n_r : ℕ, (·) (pvalue n_r) = c_r ∧ ↓𝒪 of n_a n_r.
  Definition ↓^p (p_f : P) (c_a : C_? P) : Prop := ∃ of : 𝒪, pop of = p_f ∧ ∃ n_a : ℕ, (·) (pvalue n_a) = c_a.

  Lemma preservation_prim (p_f : P) (c_a : C_? P) (c_r : C_? P) : ✓P p_f → ✓C_? ✓P c_a → ↓c^p p_f c_a c_r → ✓C_? ✓P c_r.
  Lemma progress_prim (p_f : P) (c_a : C_? P) : ✓P p_f → ✓C_? ✓P c_a → ↓^p p_f c_a → ∃ c_r : C_? P, ↓c^p p_f c_a c_r.
End FlipApplicativeStructure.
```

E-C2. Proof that Flip-RCA is an RCA.

The proof of Lemma 4.3 is a trival application of the *FreeRCA* module.

```
Module FlipRCA := FreeRelationalCombinatoryAlgebra FlipApplicativeStructure.
```

E-C3. Definition of Flip-RCA Codes with Primitive Holes

The first part of this module provides the formal definitions of codes with holes and of the operations for filling those holes used in Lemma 4.4.

```
Module FlipNCC.

  Inductive 𝒪̇ : Type := hohole | hoconst (n : ℕ) | hosucc | hobranch (n : list ℕ) (ȯ_on ȯ_off : 𝒪̇) | hoflip.
  Fixpoint fill_op (o : 𝒪) (ȯ : 𝒪̇) : 𝒪
  := match ȯ with
     | hohole ↦ o
     | hoconst n ↦ const n
     | hosucc ↦ succ
     | hobranch n ȯ_on ȯ_off ↦ on · do · else· n (fill_op o ȯ_on) (fill_op o ȯ_off)
     | hoflip ↦ flip
     end.
  Inductive Ṗ : Type := hpvalue (n : ℕ) | hpop (ȯ : 𝒪̇).
  Definition Ċ : Type := C_? Ṗ.

  Definition fill_prim (o : 𝒪) (ṗ : Ṗ) : P := match ṗ with hpvalue n ↦ pvalue n | hpop ȯ ↦ pop (fill_op o ȯ) end.
  Definition fill_lambda (fill_code : Ċ → C) {n : ℕ} : L Ċ n → L C n
```

```
:= FIX fill_lambda (hl : L Ċ n) : L C n
:= MATCH hl WITH (·) ċ ↦ (·) (fill_code ċ) | lvar _ v ↦ lvar n v | · hlf hla ↦ · (fill_lambda hlf) (fill_lambda hla) END.
FIXPOINT fill_code (o : O) (ċ : Ċ) : C := MATCH ċ WITH (·) ṗ ↦ (·) (fill_prim o ṗ) | λ˙ n hl ↦ λ˙ n (fill_lambda (fill_code o) hl) END.
```

E-C4. Proof that Flip-RCA is Extensional and Finitary

The actual proof is elided here, as it is straightforward from induction on the given proof of application. Only the formal statement of Lemma 4.4 is shown.

```
LEMMA continuity_code (ċf ċa : Ċ) (o : O) (cr : C)
: ↓c^C? ↓p^p (fill_code o ċf) (fill_code o ċa) cr
→ ∃ ċr : Ċ,
   fill_code o ċr = cr
∧ ∃ io : list (prod N N),
   Forall (λ io ↦ ↓O o (fst io) (snd io)) io
∧ ∀ o' : O, Forall (λ io ↦ ↓O o' (fst io) (snd io)) io → ↓c^C? ↓p^p (fill_code o' ċf) (fill_code o' ċa) (fill_code o' ċr).
```

E-C5. Proof that Flip-RCA Internally Negates Countable Choice

The detailed proof of Theorem 4.6 is elided here, but we provide the primary lemmas to provide some insight.

```
FIXPOINT filled_op (o : O) : Ȯ
:= MATCH o WITH const n ↦ hoconst n | succ ↦ hosucc | on · do · else· n o_on o_off ↦ hobranch n (filled_op o_on) (filled_op o_off)
| flip ↦ hoflip END.
DEFINITION filled_prim (p : P) : Ṗ := MATCH p WITH pvalue n ↦ hpvalue n | pop o ↦ hpop (filled_op o) END.
DEFINITION filled_lambda (filled_code : C → Ċ) {n : N} : L C n → L Ċ n
:= FIX filled_lambda (ℓ : L C n) : L Ċ n
:= MATCH ℓ WITH (·) c ↦ (·) (filled_code c) | lvar _ v ↦ lvar n v | · ℓf ℓa ↦ · (filled_lambda ℓf) (filled_lambda ℓa) END.
FIXPOINT filled_code (c : C) : Ċ := MATCH c WITH (·) p ↦ (·) (filled_prim p) | λ˙ n ℓ ↦ λ˙ n (filled_lambda filled_code ℓ)
END.
LEMMA fill_filled_code (o : O) (c : C) : fill_code o (filled_code c) = c.

LEMMA red_cnat (n : N) : ↓c^C? ↓p^p ↓^p (c^λ n) ((·) (pop succ)) ∧ ∀ cns : C, ↓c^C? ↓p^p (c^λ n) ((·) (pop succ)) cns → ∀ m : N,
↓c^C? ↓p^p ↓^p cns ((·) (pvalue m)) ∧ ∀ cnm : C, ↓c^C? ↓p^p cns ((·) (pvalue m)) cnm → (·) (pvalue (n + m)) = cnm.

DEFINITION cop (o : O) : C := λ˙ 0 (· ((·) ((·) (pop o))) (· (· (lvar 1 None) ((·) ((·) (pop succ)))) ((·) ((·) (pvalue 0))))).
LEMMA red_cop (o : O) (m : N) (cm : C) (φr : C → Prop) : m m cm → (∀ n : N, ↓O o m n → φr ((·) (pvalue n))) → ↓φ
↓c ↓ (cop o) cm φr.
DEFINITION hcop (ȯ : Ȯ) : Ċ := λ˙ 0 (· ((·) ((·) (hpop ȯ))) (· (· (lvar 1 None) ((·) ((·) (hpop hosucc)))) ((·) ((·) (hpvalue
0))))).
LEMMA fill_cop (o : O) (ȯ : Ȯ) : fill_code o (hcop ȯ) = cop (fill_op o ȯ).

DEFINITION Rop (o : O) : set (⇒ N (⇒ N N Ω)) := λ m n c ↦ ↓O o m n ∧ ((·) (pvalue n)) = c.
LEMMA Ropv (o : O) : √set (⇒ N (⇒ N N Ω)) (Rop o).
LEMMA Rop_total (o : O) : total N N m (Rop o) (cop o).

DEFINITION ncc : C := caxiom.
THEOREM neg_countable_choice : ∃ o : O, ⊢ (countable_choice o) (⊥ 1).
END FlipNCC.
```

E-D. CCSCA.v

E-D1. Definition of Mem-SCA

This is the formal definition of Mem-SCA in Figure 3. It uses the *FreeSCA* module for λ-terms, so the following module specifies the primitives and their behavior. There are a few differences between this definition and that in Figure 3, all for the sake of reducing metatheoretic assumptions. In particular, Σ (i.e. the set of ς s) is defined inductively as a sequence of allocation/memoization events, and ≤ is simply to be defined to be prefix. This means that if a pre-state has two predecessors, then one of the those predecessors must be a future of the other. It also means that if a location-input pair has a memoization in the current pre-state, then we can determine via *sfirst* the first point in the past where that pair was allocated in the pre-state. Any subsequent futures will have that same first state for the given entry, as proven by *sfirst_fut_eq*. This means that we can take a choice relation that is *not* future-stable, and define from it a choice relation that *is* future-stable by having every state instead the choice for a particular entry that was assigned to its first predecessor that had that entry (which might be itself). Beyond this, the proof is the same is given in Theorem 5.6.

```
MODULE MemoizingApplicativeStructure <: PrimitiveApplicativeStructure.
```

Primitives and Pre-States

```
DEFINITION L : Set := N.
INDUCTIVE P : Set := ndnat | memo | lookup (ℓ : L).

INDUCTIVE Σ : Set := sempty | sallocate (σ : Σ) (ℓ : L) (cf : C? P) | smemoize (σ : Σ) (ℓ : L) (n : N) (cr : C? P).

INDUCTIVE ≤ (σ : Σ) : Σ → Prop
:= freflΣ : ≤ σ σ
 | fallocate (σ' : Σ) (ℓ : L) (cf : C? P) : ≤ σ σ' → ≤ σ (sallocate σ' ℓ cf)
 | fmemoize (σ' : Σ) (ℓ : L) (n : N) (cr : C? P) : ≤ σ σ' → ≤ σ (smemoize σ' ℓ n cr).

LEMMA ftrans' (σ σ' σ'' : Σ) : ≤ σ σ' → ≤ σ' σ'' → ≤ σ σ''.

LEMMA flinear {s1 s2 σ' : Σ} : ≤ s1 σ' → ≤ s2 σ' → ≤ s1 s2 ∨ ≤ s2 s1.

INDUCTIVE Allocated (σ : Σ) (ℓ : L) (cf : C? P) : Prop := aallocate (sa : Σ) : ≤ (sallocate sa ℓ cf) σ → Allocated σ ℓ cf.
INDUCTIVE Memoized (σ : Σ) (ℓ : L) (n : N) (cr : C? P) : Prop := mmemoize (sm : Σ) : ≤ (smemoize sm ℓ n cr) σ →
Memoized σ ℓ n cr.
INDUCTIVE √P (σ : Σ) : P → Prop
:= pndv : √P σ ndnat
 | pallocv : √P σ memo
 | plookupv (n : N) (cf : C? P) : Allocated σ n cf → √P σ (lookup n).

LEMMA allocated_fut (σ σ' : Σ) (ℓ : L) (cf : C? P) : ≤ σ σ' → Allocated σ ℓ cf → Allocated σ' ℓ cf.
LEMMA memoized_fut (σ σ' : Σ) (ℓ : L) (n : N) (cr : C? P) : ≤ σ σ' → Memoized σ ℓ n cr → Memoized σ' ℓ n cr.
```

Lemma *primv_fut'* $(\sigma\ \sigma' : \Sigma)\ (p : P) : \leq \sigma\ \sigma' \to \checkmark_P\ \sigma\ p \to \checkmark_P\ \sigma'\ p.$
Lemma *pcodev_fut'* $(\sigma\ \sigma' : \Sigma)\ (c : C_?\ P) : \leq \sigma\ \sigma' \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c \to \checkmark_{C_?}\ (\checkmark_P\ \sigma')\ c.$

Fixpoint *cchurch* $\{P : \mathbf{Set}\}\ (n : \mathbb{N}) : C_?\ P$
$:=$ match n with $0 \mapsto \lambda^{\cdot}\ 1\ (lvar\ 2\ (Some\ None)) \mid S\ n \mapsto \lambda^{\cdot}\ 1\ (\cdot\ (lvar\ 2\ None)\ (\cdot\ (\cdot\ ((\cdot)\ (cchurch\ n))\ (lvar\ 2\ None))\ (lvar\ 2\ (Some\ None))))$ end.

Application within a Pre-State

These define *frozen* reduction within a pre-state.

Inductive $\downarrow^p_\varsigma ?\ (\sigma : \Sigma) : P \to C_?\ P \to C_?\ P \to$ Prop
$:=$ *rpfnd* $(c_a : C_?\ P)\ (n : \mathbb{N}) : \downarrow^p_\varsigma ?\ \sigma$ *ndnat* $c_a\ (cchurch\ n)$
| *rpfalloc* $(c_f : C_?\ P)\ (\ell : \mathcal{L}) : Allocated\ \sigma\ \ell\ c_f \to \downarrow^p_\varsigma ?\ \sigma$ *memo* $c_f\ ((\cdot)\ (\mathsf{lookup}\ \ell))$
| *rpflookup* $(\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_r : C_?\ P) : Memoized\ \sigma\ \ell\ n\ c_r \to \downarrow^p_\varsigma ?\ \sigma\ (\mathsf{lookup}\ \ell)\ (cchurch\ n)\ c_r.$
Inductive $\downarrow^\lambda_\varsigma\ (\downarrow^{C_?}_\varsigma : C_?\ P \to C_?\ P \to C_?\ P \to$ Prop$) : L\ (C_?\ P)\ 0 \to C_?\ P \to$ Prop
$:=$ *rlfcode* $(c : C_?\ P) : \downarrow^\lambda_\varsigma \downarrow^{C_?}_\varsigma\ ((\cdot)\ c)\ c$
| *rlfapp* $(lf\ la : L\ (C_?\ P)\ 0)\ (c_f\ c_a\ c_r : C_?\ P) : \downarrow^\lambda_\varsigma \downarrow^{C_?}_\varsigma\ lf\ c_f \to \downarrow^\lambda_\varsigma \downarrow^{C_?}_\varsigma\ la\ c_a \to \downarrow^{C_?}_\varsigma\ c_f\ c_a\ c_r \to \downarrow^\lambda_\varsigma \downarrow^{C_?}_\varsigma\ (\cdot\ lf\ la)\ c_r.$
Inductive $\downarrow^{C_?}_\varsigma\ (\sigma : \Sigma) : C_?\ P \to C_?\ P \to C_?\ P \to$ Prop
$:=$ *rcprim* $(p_f : P)\ (c_a : C_?\ P)\ (c_r : C_?\ P) : \downarrow^p_\varsigma ?\ \sigma\ p_f\ c_a\ c_r \to \downarrow^{C_?}_\varsigma\ \sigma\ ((\cdot)\ p_f)\ c_a\ c_r$
| *rclam0* $(lb : L\ (C_?\ P)\ 1)\ (c_a\ c_r : C_?\ P) : \downarrow^\lambda_\varsigma (\downarrow^{C_?}_\varsigma\ \sigma)\ (lsubst\ c_a\ lb)\ c_r \to \downarrow^{C_?}_\varsigma\ \sigma\ (\lambda^{\cdot}\ 0\ lb)\ c_a\ c_r$
| *rclamS* $(n : \mathbb{N})\ (lb : L\ (C_?\ P)\ (S\ (S\ n)))\ (c_a : C_?\ P) : \downarrow^{C_?}_\varsigma\ \sigma\ (\lambda^{\cdot}\ (S\ n)\ lb)\ c_a\ (\lambda^{\cdot}\ n\ (lsubst\ c_a\ lb)).$

Lemma *red_lambda_frozen_fut* $(\downarrow^{C_?}_\varsigma\ \Downarrow^{C_?}_\varsigma : C_?\ P \to C_?\ P \to C_?\ P \to$ Prop$)\ (\ell : L\ (C_?\ P)\ 0)\ (c_r : C_?\ P) : (\forall\ c_f\ c_a\ c_r : C_?\ P, \downarrow^{C_?}_\varsigma\ c_f\ c_a\ c_r \to \Downarrow^{C_?}_\varsigma\ c_f\ c_a\ c_r) \to \downarrow^\lambda_\varsigma \downarrow^{C_?}_\varsigma\ \ell\ c_r \to \downarrow^\lambda_\varsigma \Downarrow^{C_?}_\varsigma\ \ell\ c_r.$
Lemma *red_prim_frozen_fut* $(\sigma\ \sigma' : \Sigma)\ (p_f : P)\ (c_a : C_?\ P)\ (c_r : C_?\ P) : \leq \sigma\ \sigma' \to \downarrow^p_\varsigma ?\ \sigma\ p_f\ c_a\ c_r \to \downarrow^p_\varsigma ?\ \sigma'\ p_f\ c_a\ c_r.$
Lemma *red_code_frozen_fut* $(\sigma\ \sigma' : \Sigma)\ (c_f\ c_a\ c_r : C_?\ P) : \leq \sigma\ \sigma' \to \downarrow^{C_?}_\varsigma\ \sigma\ c_f\ c_a\ c_r \to \downarrow^{C_?}_\varsigma\ \sigma'\ c_f\ c_a\ c_r.$

Inductive *UnAllocated* $: \Sigma \to \mathcal{L} \to$ Prop
$:=$ *uaempty* $(lf : \mathcal{L}) : UnAllocated\ sempty\ lf$
| *uaallocate* $(lf : \mathcal{L})\ (\sigma : \Sigma)\ (\ell : \mathcal{L})\ (c_f : C_?\ P) : UnAllocated\ \sigma\ lf \to (lf = \ell \to$ False$) \to UnAllocated\ (sallocate\ \sigma\ \ell\ c_f)\ lf$
| *uamemoized* $(lf : \mathcal{L})\ (\sigma : \Sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_r : C_?\ P) : UnAllocated\ \sigma\ lf \to UnAllocated\ (smemoize\ \sigma\ \ell\ n\ c_r)\ lf.$
Inductive *UnMemoized* $: \Sigma \to \mathcal{L} \to \mathbb{N} \to$ Prop
$:=$ *umempty* $(lf : \mathcal{L})\ (nf : \mathbb{N}) : UnMemoized\ sempty\ lf\ nf$
| *umallocate* $(lf : \mathcal{L})\ (nf : \mathbb{N})\ (\sigma : \Sigma)\ (\ell : \mathcal{L})\ (c_f : C_?\ P) : UnMemoized\ \sigma\ lf\ nf \to UnMemoized\ (sallocate\ \sigma\ \ell\ c_f)\ lf\ nf$
| *ummemoized* $(lf : \mathcal{L})\ (nf : \mathbb{N})\ (\sigma : \Sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_r : C_?\ P) : UnMemoized\ \sigma\ lf\ nf \to (lf = \ell \to nf = n \to$ False$) \to UnMemoized\ (smemoize\ \sigma\ \ell\ n\ c_r)\ lf\ nf.$

Lemma *memoized_unmemoized_false* $(\sigma : \Sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_r : C_?\ P) : Memoized\ \sigma\ \ell\ n\ c_r \to UnMemoized\ \sigma\ \ell\ n \to$ False.

States

Rather than defining state validity by the behaviors we need of states, as in Figure 3, we define state validity inductively and prove that it implies the necessary behaviors.

Inductive $\checkmark_\Sigma : \Sigma \to$ Prop
$:=$ *semptyv* $: \checkmark_\Sigma\ sempty$
| *sallocatev* $(\sigma : \Sigma)\ (\ell : \mathcal{L})\ (c_f : C_?\ P) : \checkmark_\Sigma\ \sigma \to UnAllocated\ \sigma\ \ell \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_f \to \checkmark_\Sigma\ (sallocate\ \sigma\ \ell\ c_f)$
| *smemoizev* $(\sigma : \Sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_r : C_?\ P)\ (c_f : C_?\ P) : \checkmark_\Sigma\ \sigma \to Allocated\ \sigma\ \ell\ c_f \to UnMemoized\ \sigma\ \ell\ n \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_r \to \downarrow^{C_?}_\varsigma\ \sigma\ c_f\ (cchurch\ n)\ c_r \to \checkmark_\Sigma\ (smemoize\ \sigma\ \ell\ n\ c_r).$
Lemma *statev_fut* $(\sigma\ \sigma' : \Sigma) : \leq \sigma\ \sigma' \to \checkmark_\Sigma\ \sigma' \to \checkmark_\Sigma\ \sigma.$
Lemma *allocated_det* $(\sigma : \Sigma)\ (sv : \checkmark_\Sigma\ \sigma)\ (\ell : \mathcal{L})\ (c_f\ cf' : C_?\ P) : Allocated\ \sigma\ \ell\ c_f \to Allocated\ \sigma\ \ell\ cf' \to c_f = cf'.$
Lemma *memoized_det* $(\sigma : \Sigma)\ (sv : \checkmark_\Sigma\ \sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_r\ cr' : C_?\ P) : Memoized\ \sigma\ \ell\ n\ c_r \to Memoized\ \sigma\ \ell\ n\ cr' \to c_r = cr'.$
Lemma *allocated_valid* $(\sigma : \Sigma)\ (sv : \checkmark_\Sigma\ \sigma)\ (\ell : \mathcal{L})\ (c_f : C_?\ P) : Allocated\ \sigma\ \ell\ c_f \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_f.$
Lemma *memoized_valid* $(\sigma : \Sigma)\ (sv : \checkmark_\Sigma\ \sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_r : C_?\ P) : Memoized\ \sigma\ \ell\ n\ c_r \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_r.$
Lemma *memoized_allocated* $(\sigma : \Sigma)\ (sv : \checkmark_\Sigma\ \sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_r : C_?\ P) : Memoized\ \sigma\ \ell\ n\ c_r \to \exists\ c_f : C_?\ P, Allocated\ \sigma\ \ell\ c_f.$
Lemma *memoized_red* $(\sigma : \Sigma)\ (sv : \checkmark_\Sigma\ \sigma)\ (\ell : \mathcal{L})\ (c_f : C_?\ P)\ (n : \mathbb{N})\ (c_r : C_?\ P) : Allocated\ \sigma\ \ell\ c_f \to Memoized\ \sigma\ \ell\ n\ c_r \to \downarrow^{C_?}_\varsigma\ \sigma\ c_f\ (cchurch\ n)\ c_r.$

Lemma *sinhabited* $: \exists\ \sigma : \Sigma, \checkmark_\Sigma\ \sigma.$
Lemma *frefl* $(\sigma : \Sigma) : \checkmark_\Sigma\ \sigma \to \leq \sigma\ \sigma.$
Lemma *ftrans* $(\sigma\ \sigma'\ \sigma'' : \Sigma) : \checkmark_\Sigma\ \sigma \to \checkmark_\Sigma\ \sigma' \to \checkmark_\Sigma\ \sigma'' \to \leq \sigma\ \sigma' \to \leq \sigma'\ \sigma'' \to \leq \sigma\ \sigma''.$
Lemma *primv_fut* $(\sigma\ \sigma' : \Sigma)\ (p : P) : \checkmark_\Sigma\ \sigma \to \checkmark_\Sigma\ \sigma' \to \leq \sigma\ \sigma' \to \checkmark_P\ \sigma\ p \to \checkmark_P\ \sigma'\ p.$
Lemma *pcodev_fut* $(\sigma\ \sigma' : \Sigma)\ (c : C_?\ P) : \checkmark_\Sigma\ \sigma \to \checkmark_\Sigma\ \sigma' \to \leq \sigma\ \sigma' \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c \to \checkmark_{C_?}\ (\checkmark_P\ \sigma')\ c.$

Application and Termination with States

This defines the application relation that *does* mutate pre-state.

Definition $\downarrow^p_c\ (\sigma : \Sigma)\ (p_f : P)\ (c_a : C_?\ P)\ (\sigma' : \Sigma)\ (c_r : C_?\ P) :$ Prop $:= \leq \sigma\ \sigma' \wedge \checkmark_\Sigma\ \sigma' \wedge \downarrow^p_\varsigma ?\ \sigma'\ p_f\ c_a\ c_r.$
Inductive $\downarrow^p\ (\sigma : \Sigma) : P \to C_?\ P \to$ Prop
$:=$ *tpfnd* $(c_a : C_?\ P) : \downarrow^p\ \sigma$ *ndnat* c_a
| *tpfalloc* $(c_f : C_?\ P) : \downarrow^p\ \sigma$ *memo* c_f
| *tpflookup* $(\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_f : C_?\ P) : Allocated\ \sigma\ \ell\ c_f \to \downarrow^{C_?}\ \downarrow^p_c \downarrow^p\ \sigma\ c_f\ (cchurch\ n) \to \downarrow^p\ \sigma\ (\mathsf{lookup}\ \ell)\ (cchurch\ n).$

Progress

Lemma *preservation_prim* $(\sigma : \Sigma)\ (p_f : P)\ (c_a : C_?\ P)\ (\sigma' : \Sigma)\ (c_r : C_?\ P) : \checkmark_\Sigma\ \sigma \to \checkmark_P\ \sigma\ p_f \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_a \to \downarrow^p_c\ \sigma\ p_f\ c_a\ \sigma'\ c_r \to \checkmark_\Sigma\ \sigma' \wedge \leq \sigma\ \sigma' \wedge \checkmark_{C_?}\ (\checkmark_P\ \sigma')\ c_r.$

Lemma *new* $(\sigma : \Sigma) : (\exists\ \ell : \mathcal{L}, \forall\ \ell' : \mathcal{L}, \ell \leq \ell' \to UnAllocated\ \sigma\ \ell').$
Lemma *memoized* $(\sigma : \Sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N}) : (\exists\ c_r : C_?\ P, Memoized\ \sigma\ \ell\ n\ c_r) \vee (UnMemoized\ \sigma\ \ell\ n).$

Lemma *red_code_freeze* $(\sigma : \Sigma)\ (c_f\ c_a : C_?\ P)\ (\sigma' : \Sigma)\ (c_r : C_?\ P) : \checkmark_\Sigma\ \sigma \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_f \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_a \to \downarrow_c^{C_?}\ \downarrow_c^p\ \sigma\ c_f\ c_a\ \sigma'\ c_r \to \downarrow_\varsigma^{C_?}\ \sigma'\ c_f\ c_a\ c_r$.

Lemma *red_code_thaw* $(\sigma : \Sigma)\ (c_f\ c_a : C_?\ P)\ (c_r : C_?\ P) : \checkmark_\Sigma\ \sigma \to \downarrow_\varsigma^{C_?}\ \sigma\ c_f\ c_a\ c_r \to \downarrow_c^{C_?}\ \downarrow_c^p\ \sigma\ c_f\ c_a\ \sigma\ c_r$.

Lemma *progress_prim* $(\sigma : \Sigma)\ (p_f : P)\ (c_a : C_?\ P) : \checkmark_\Sigma\ \sigma \to \checkmark_P\ \sigma\ p_f \to \checkmark_{C_?}\ (\checkmark_P\ \sigma)\ c_a \to \downarrow^p\ \sigma\ p_f\ c_a \to \exists\ \sigma' : \Sigma, \exists\ c_r : C_?\ P, \downarrow_c^p\ \sigma\ p_f\ c_a\ \sigma'\ c_r$.

End *MemoizingApplicativeStructure*.

E-D2. Proof that Mem-SCA is an SCA.

The proof of Lemma 5.4 is a trival application of the *FreeSCA* module.

Module *MemoizingSCA* := *FreeStatefulCombinatoryAlgebra MemoizingApplicativeStructure*.

E-D3. Proof that Mem-SCA Internally Models Countable Choice

The detailed proof of Theorem 5.6 is elided here, but we provide the primary lemmas to provide some insight.

Module *MemoizingCC*.

Here we assume the axiom of countable choice in the metatheory, asserting without proof that the set of states Σ and the set of codes C are both countable, and that $\checkmark_\Sigma$, $\leq$, and *Memoized* are each recognizable predicates so that the subset of states and codes satisfying them is also countable.

Axiom *axiom_of_countable_choice* : $\forall\ I$: **Set**, $\forall\ \checkmark_I : I \to$ Prop, $\forall\ \sigma : \Sigma, \forall\ \ell : \mathcal{L}, \forall\ R : \mathbb{N} \to I \to \Sigma \to C \to$ Prop, $(\forall\ n : \mathbb{N}, \forall\ \sigma' : \Sigma, \forall\ c : C, \checkmark_\Sigma\ \sigma' \to \leq\ \sigma\ \sigma' \to \mathit{Memoized}\ \sigma'\ \ell\ n\ c \to \exists\ i : I, \checkmark_I\ i \wedge R\ n\ i\ \sigma'\ c) \to \exists\ S : \mathbb{N} \to I \to \Sigma \to C \to$ Prop, $(\forall\ n : \mathbb{N}, \forall\ i : I, \forall\ \sigma' : \Sigma, \forall\ c : C, S\ n\ i\ \sigma'\ c \to R\ n\ i\ \sigma'\ c) \wedge (\forall\ n : \mathbb{N}, \forall\ i\ i' : I, \forall\ \sigma' : \Sigma, \forall\ c : C, S\ n\ i\ \sigma'\ c \to S\ n\ i'\ \sigma'\ c \to i = i') \wedge \forall\ n : \mathbb{N}, \forall\ \sigma' : \Sigma, \forall\ c : C, \checkmark_\Sigma\ \sigma' \to \leq\ \sigma\ \sigma' \to \mathit{Memoized}\ \sigma'\ \ell\ n\ c \to \exists\ i : I, \checkmark_I\ i \wedge S\ n\ i\ \sigma'\ c$.

First predecessor with a given entry

Fixpoint *sfirst* $(\ell : \mathcal{L})\ (n : \mathbb{N})\ (\sigma : \Sigma) : \Sigma$
:= match σ with
| *sempty* $\mapsto$ *sempty*
| *sallocate* $\sigma\ \ell'\ c_f \mapsto$ *sfirst* $\ell\ n\ \sigma$
| *smemoize* $\sigma\ \ell'\ n'\ c_r \mapsto$ if *eq_dec* $\ell\ \ell'$ then if *eq_dec* $n\ n'$ then *smemoize* $\sigma\ \ell'\ n'\ c_r$ else *sfirst* $\ell\ n\ \sigma$ else *sfirst* $\ell\ n\ \sigma$
end.

Lemma *smemoize_sfirst_fut* $(\sigma\ \sigma' : \Sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N})\ (c_r : C) : \leq (\mathit{smemoize}\ \sigma\ \ell\ n\ c_r)\ \sigma' \to \leq (\mathit{smemoize}\ \sigma\ \ell\ n\ c_r)\ (\mathit{sfirst}\ \ell\ n\ \sigma')$.

Lemma *sfirst_memoized* $(\ell : \mathcal{L})\ (n : \mathbb{N})\ (\sigma : \Sigma)\ (c_r : C) : \mathit{Memoized}\ \sigma\ \ell\ n\ c_r \to \mathit{Memoized}\ (\mathit{sfirst}\ \ell\ n\ \sigma)\ \ell\ n\ c_r$.

Lemma *sfirst_fut_eq* $(\ell : \mathcal{L})\ (n : \mathbb{N})\ (\sigma\ \sigma' : \Sigma)\ (c_r : C) : \checkmark_\Sigma\ \sigma' \to \leq\ \sigma\ \sigma' \to \mathit{Memoized}\ \sigma\ \ell\ n\ c_r \to \mathit{sfirst}\ \ell\ n\ \sigma = \mathit{sfirst}\ \ell\ n\ \sigma'$.

Lemma *sfirst_fut* $(\ell : \mathcal{L})\ (n : \mathbb{N})\ (\sigma : \Sigma) : \leq (\mathit{sfirst}\ \ell\ n\ \sigma)\ \sigma$.

Lemma *sfirst_unmemoized* $(\sigma : \Sigma)\ (\ell : \mathcal{L})\ (n : \mathbb{N})\ (\sigma' : \Sigma)\ (c_r : C) : \mathit{UnMemoized}\ \sigma\ \ell\ n \to \mathit{Memoized}\ \sigma'\ \ell\ n\ c_r \to \leq\ \sigma\ \sigma' \to \leq\ \sigma\ (\mathit{sfirst}\ \ell\ n\ \sigma')$.

Countable Choice

Definition *cc* : $C := c_\lambda\ 0\ (\mathit{ecode}\ ((\cdot)\ \mathit{memo}))$.
Theorem *countable_choice* $(o : \mathcal{O}) : \checkmark_\mathcal{O}\ o \to \vdash (\top\ \mathbf{1})\ (\mathit{countable\_choice}\ o)$.

End *MemoizingCC*.