Computing in Cantor's Paradise With $\lambda_{\mathbf{ZFC}}$

Neil Toronto and Jay McCarthy
neil.toronto@gmail.com and jay@cs.byu.edu

PLT @ Brigham Young University, Provo, Utah, USA

Abstract. Applied mathematicians increasingly use computers to answer mathematical questions. We want to provide them domain-specific languages. The languages should have exact meanings and computational meanings. Some proof assistants can encode exact mathematics and extract programs, but formalizing the required theorems can take years.
As an alternative, we develop λ_{ZFC}, a lambda calculus that contains infinite sets as values, in which to express exact mathematics and gradually change infinite calculations to computable ones. We define it as a conservative extension of set theory, and prove that most contemporary theorems apply directly to λ_{ZFC} terms.
We demonstrate λ_{ZFC}'s expressiveness by coding up the real numbers, arithmetic and limits. We demonstrate that it makes deriving computational meaning easier by defining a monad in it for expressing limits, and using standard topological theorems to derive a computable replacement.

Keywords: Lambda Calculus, Set Theory, Semantics

No one shall expel us from the Paradise that Cantor has created.
David Hilbert

1 Introduction

Georg Cantor first proved some of the surprising consequences of assuming infinite sets exist. David Hilbert passionately defended Cantor's set theory as a mathematical foundation, coining the term "Cantor's Paradise" to describe the universe of transfinite sets in which most mathematics now takes place.

The calculations done in Cantor's Paradise range from computable to unimaginably uncomputable. Still, its inhabitants increasingly use computers to answer questions. We want to make domain-specific languages (DSLs) for writing these questions, with implementations that compute exact and approximate answers.

Such a DSL should have two meanings: an exact mathematical semantics, and an approximate computational one. A traditional, denotational approach is to give the exact as a transformation to first-order set theory, and because set theory is unlike any intended implementation language, the approximate as a transformation to a lambda calculus. However, deriving approximations while switching target languages is rife with opportunities to commit errors.

A more certain way is to define the exact semantics in a proof assistant like HOL [11] or Coq [5], prove theorems, and extract programs. The type systems confer an advantage: if the right theorems are proved, the programs are correct.

Unfortunately, reformulating and re-proving theorems in such an exacting way causes significant delays. For example, half of Joe Hurd's 2002 dissertation on probabilistic algorithms [9] is devoted to formalizing early-1900s measure theory in HOL. Our work in Bayesian inference would require at least three times as much formalization, even given the work we could build on.

Some middle ground is clearly needed: something between the traditional, error-prone way and the slow, absolutely certain way.

Instead of using a typed, higher-order logic, suppose we defined, in first-order set theory, an untyped lambda calculus that contained infinite sets and operations on them. We could interpret DSL terms exactly as uncomputable programs in this lambda calculus. But instead of redoing a century of work to extract programs that compute approximations, we could directly reuse first-order theorems to derive them from the uncomputable programs.

Conversely, set theory, which lacks lambdas and general recursion, is an awkward target language for a semantics that is intended to be implemented. Suppose we extended set theory with untyped lambdas (as objects, not quantifiers). We could still interpret DSL terms as operations on infinite objects. But instead of leaping from infinite sets and operations on them to implementations, we could replace those operations with computable approximations piece at a time.

If we had a lambda calculus with infinite sets as values, we could approach computability from above in a principled way, gradually changing programs for Cantor's Paradise until they can be implemented in Church's Purgatory.

We define that lambda calculus, λ_{ZFC}, and a call-by-value, big-step reduction semantics. To show that it is expressive enough, we code up the real numbers, arithmetic and limits, following standard analysis. To show that it simplifies language design, we define the uncomputable limit monad in λ_{ZFC}, and derive a computable, directly implementable replacement monad by applying standard topological theorems. When certain proof obligations are met, the output of programs that use the computable monad converge to the same values as the output of programs that use the uncomputable monad but are otherwise identical.

1.1 Language Tower and Terminology

λ_{ZFC}'s metalanguage is **first-order set theory**: first-order logic with equality extended with ZFC, or the Zermelo Fraenkel axioms and Choice (equivalently well-ordering). We also assume the existence of an inaccessible cardinal. Section 2 reviews the axioms, which λ_{ZFC}'s primitives are derived from.

To help ensure λ_{ZFC}'s definition conservatively extends set theory, we encode its terms as sets. For example, $\langle t_{\mathcal{P}}, \mathbb{R} \rangle$ is the encoding of "the powerset of the reals" as a pair, where $\langle x, y \rangle = \{\{x\}, \{x, y\}\}$ for any sets x and y.

λ_{ZFC}'s semantics reduces terms to terms; e.g. $\langle t_{\mathcal{P}}, \mathbb{R} \rangle$ reduces to the actual powerset of $\mathbb{R}$. Thus, λ_{ZFC} contains infinite terms. Infinitary languages are useful and definable: the infinitary lambda calculus [10] is an example, and Aczel's broadly used work [2] on inductive sets treats infinite inference rules explicitly.

For convenience, we define a language λ^{-}_{ZFC} of finite terms and a function $\mathcal{F}[\![\cdot]\!]$ from λ^{-}_{ZFC} to λ_{ZFC}. We can then write $\mathcal{P}\ \mathbb{R}$, meaning $\mathcal{F}[\![\mathcal{P}\ \mathbb{R}]\!] = \langle t_{\mathcal{P}}, \mathbb{R} \rangle$.

Semantic functions like $\mathcal{F}[\![\cdot]\!]$ and the interpretation of BNF grammars are defined in set theory's metalanguage, or the *meta*-metalanguage. Distinguishing metalanguages helps avoid paradoxes of definition such as Berry's paradox, which are particularly easy to stumble onto when dealing with infinities.

We write $\lambda^{-}_{\mathrm{ZFC}}$ terms in sans serif font, and the metalanguage and meta-metalanguage in *math font*. We write common keywords in **bold** and invented keywords in ***bold italics***. We abbreviate proofs for space.

2 Metalanguage: First-Order Set Theory

We assume readers are familiar with classical first-order logic with equality and its inference rules, but not set theory. Hrbacek and Jech [8] is a fine introduction.

Set theory extends classical first-order logic with equality, which distinguishes between truth-valued formulas ϕ and object-valued terms x. Set theory allows only sets as objects, and quantifiers like '$\forall$' may range only over sets.

We define predicates and functions using ':='; for example, $nand(\phi_1, \phi_2) := \neg(\phi_1 \wedge \phi_2)$. They must be nonrecursive so they can be exhaustively applied. They are therefore **conservative extensions**: they do not prove more theorems.

To develop set theory, we make **proper extensions**, which prove more theorems, by adding symbols and axioms to first-order logic. For example, we first add '$\varnothing$' and '$\in$', and the **empty set axiom** $\forall\, x.\, x \notin \varnothing$.

We use ':$\equiv$' to define syntax; e.g. $\forall\, x \in A.\, P(x) \;:\equiv\; \forall\, x.\, (x \in A \Rightarrow P(x))$, where predicate application $P(x)$ represents a formula that may depend on x. We allow recursion in meta-metalanguage definitions if substitution terminates, so $\forall\, x_1\, x_2 \ldots x_n.\, \phi :\equiv \forall\, x_1.\, \forall\, x_2 \ldots x_n.\, \phi$ can bind any number of names.

We already have Axiom 0 (empty set). Now for the rest.

Axiom 1 (extensionality). Define $A \subseteq B := \forall\, x \in A.\, x \in B$ and assume $\forall\, A\, B.\, (A \subseteq B \wedge B \subseteq A \Rightarrow A = B)$; i.e. $A = B$ if they mutually are subsets. □

The converse follows from substituting A for B or B for A.

Axiom 2 (foundation). Define $A \not\cap B := \forall\, x.\, (x \in A \Rightarrow x \notin B)$ ("A and B are disjoint") and assume $\forall\, A.\, (A = \varnothing) \vee \exists\, x \in A.\, x \not\cap A$. □

Foundation implies that the following nondeterministic procedure always terminates: If input $A = \varnothing$, return A; otherwise restart with any $A' \in A$.

Thus, sets are roots of trees in which every upward path is unbounded but finite. Foundation is analogous to "all data constructors are strict."

Axiom 3 (powerset). Add '$\mathcal{P}$' and assume $\forall\, A\, x.\, (x \in \mathcal{P}(A) \iff x \subseteq A)$. □

A **hereditarily finite** set is finite and has only hereditarily finite members. Each such set first appears in some $\mathcal{P}(\mathcal{P}(...\mathcal{P}(\varnothing)...))$. For example, after $\{x, ...\}$ (literal set syntax) is defined, $\{\varnothing\} \in \mathcal{P}(\mathcal{P}(\varnothing))$. $\{\mathbb{R}\}$ is not hereditarily finite.

Axiom 4 (union). Add '$\bigcup$' ("big" union) and assume $\forall\, A\, x.\, (x \in \bigcup A \iff \exists\, y.\, x \in y \wedge y \in A)$. □

After $\{x, ...\}$ is defined, $\bigcup\{\{x, y\}, \{y, z\}\} = \{x, y, z\}$. Also, '$\bigcup$' can extract the object in a singleton set: if $A = \{x\}$, then $x = \bigcup A$.

Axiom 5 (replacement schema). A binary predicate R can act as a function if $\forall\, x \in A.\, \exists!\, y.\, R(x, y)$, where '$\exists!$' means unique existence. We cannot quantify over predicates in first-order logic, but we can assume, for each such definable R, that $\forall\, y.\, (y \in \{y' \mid x \in A \wedge R(x, y')\} \iff \exists\, x \in A.\, R(x, y))$. Roughly, treating R as a function, if R's domain is a set, its image (range) is also a set. □

A **schema** represents countably many axioms. If $R(n, m) \iff m = n + 1$, for example, then $\{m \mid n \in \mathbb{N} \wedge R(n, m)\}$ increments the natural numbers.

Define $\{F(x) \mid x \in A\} :\equiv \{y \mid x \in A \wedge y = F(x)\}$, analogous to `map F A`, for **functional replacement**. Now $\{n + 1 \mid n \in \mathbb{N}\}$ increments the naturals.

It seems replacement should be *defined* functionally, but predicates allow powerful nonconstructivism. Suppose $Q(y)$ for exactly one y. The **description operator** $\iota\, y.\, Q(y) :\equiv \bigcup\{y \mid x \in \mathcal{P}(\varnothing) \wedge Q(y)\}$ finds "the y such that $Q(y)$."

From the six axioms so far, we can define $A \cup B$ (binary union), $\{x, ...\}$ (literal finite sets), $\langle x, y, z, ...\rangle$ (ordered pairs and lists), $\{x \in A \mid Q(x)\}$ (bounded selection), $A \backslash B$ (relative complement), $\bigcap A$ ("big" intersection), $\bigcup_{x \in A} F(x)$ (indexed union), $A \times B$ (cartesian product), and $A \to B$ (total function spaces). For details, we recommend Paulson's remarkably lucid development in HOL [16].

2.1 The Gateway to Cantor's Paradise: Infinity

From the six axioms so far, we cannot construct a set that is closed under unboundedly many operations, such as the language of a recursive grammar.

Example 1 (interpreting a grammar). We want to interpret $z ::= \varnothing \mid \langle \varnothing, z\rangle$. It should mean the least fixpoint of a function F_z, which, given a subset of z's language, returns a larger subset. To define F_z, replace '$\mid$' with '$\cup$', the terminal $\varnothing$ with $\{\varnothing\}$, and the rule $\langle \varnothing, z\rangle$ with functional replacement:

$$F_z(Z) \;:=\; \{\varnothing\} \cup \{\langle \varnothing, z\rangle \mid z \in Z\} \tag{1}$$

We could define $Z(0) := \varnothing$, then $Z(1) := F_z(Z(0)) = \{\varnothing, \langle \varnothing, \varnothing\rangle\}$, then $Z(2) = F_z(Z(1)) = \{\varnothing, \langle \varnothing, \varnothing\rangle, \langle \varnothing, \varnothing, \varnothing\rangle\}$, and so on. The language should be the union of all the $Z(n)$, but we cannot construct it without a set of all n. ♢

We follow Von Neumann, defining $0 := \varnothing$ as the **first ordinal number** and $s(n) := n \cup \{n\}$ to generate **successor ordinals**. Then $1 := s(0) = \{0\}$, $2 := s(1) = \{0, 1\}$, and $3 := s(2) = \{0, 1, 2\}$, and so on. The set of such numbers is the language of $n ::= 0 \mid s(n)$, which should be the least fixpoint of $F_n(N) := \{0\} \cup \{s(n) \mid n \in N\}$, similar to (1). We must assume some fixpoint exists.

Axiom 6 (infinity). $\exists\, I.\, I = F_n(I)$. □

I is a bounding set, so it may contain more than just finite ordinals. But F_n is monotone in I, so by the Knaster-Tarksi theorem (suitably restricted [15]),

$$\omega \;:=\; \bigcap\{N \subseteq I \mid N = F_n(N)\} \tag{2}$$

is the least fixpoint of F_n: the finite ordinals, a model of the natural numbers.

Example 2 (interpreting a grammar). We build the language of z recursively:

$$\begin{aligned} Z(0) &= \varnothing & \qquad Z(\omega) &= \bigcup_{n\in\omega} Z(n) \\ Z(s(n)) &= F_z(Z(n)),\ n \in \omega \end{aligned} \tag{3}$$

By induction, $Z(n)$ exists for every $n \in \omega$; therefore $Z(\omega)$ exists, so (3) is a conservative extension of set theory. It is not hard to prove (by induction) that $Z(\omega)$ is the set of all finite lists of $\varnothing$, and that it is the least fixpoint of F_z. $\diamondsuit$

Similarly to building the language $Z(\omega)$ of z in (3), we can build the set $\mathcal{V}(\omega)$ of all hereditarily finite sets (see Axiom 3) by iterating $\mathcal{P}$ instead of F_z:

$$\begin{aligned} \mathcal{V}(0) &= \varnothing & \qquad \mathcal{V}(\omega) &= \bigcup_{n\in\omega} \mathcal{V}(n) \\ \mathcal{V}(s(n)) &= \mathcal{P}(\mathcal{V}(n)),\ n \in \omega \end{aligned} \tag{4}$$

The set ω is not just a model of the natural numbers. It is also a number itself: the **first countable ordinal**. Indeed, ω is strikingly similar to every finite ordinal in two ways. First, it is defined as the set of its predecessors. Second, it has a successor $s(\omega) = \omega \cup \{\omega\}$. (Imagine it as $\{0, 1, 2, ..., \omega\}$.) Unlike finite, nonzero ordinals, ω has no *immediate* predecessor—it is a **limit ordinal**.

More limit ordinals allow iterating $\mathcal{P}$ further. It is not hard to build $\omega + \omega$, ω^2 and ω^ω as least fixpoints. The **Von Neumann hierarchy** generalizes (4):

$$\begin{aligned} \mathcal{V}(0) &= \varnothing & \qquad \mathcal{V}(\beta) &= \bigcup_{\alpha\in\beta} \mathcal{V}(\alpha),\ \text{limit ordinal } \beta \\ \mathcal{V}(s(\alpha)) &= \mathcal{P}(V(\alpha)),\ \text{ordinal } \alpha \end{aligned} \tag{5}$$

It is a theorem of ZFC that every set first appears in $\mathcal{V}(\alpha)$ for some ordinal α.

Equations (3,4,5) demonstrate **transfinite recursion**, set theory's unfold: defining a function V on ordinals, with $V(\beta)$ in terms of $V(\alpha)$ for every $\alpha \in \beta$.

2.2 Every Set Can Be Sequenced: Well-Ordering

A **sequence** is a total function from an ordinal to a codomain; e.g. $f \in 3 \to A$ is a length-3 sequence of A's elements. (An ordinal is comprised of its predecessors, so $3 = \{0, 1, 2\}$.) A **well-order** of A is a bijective sequence of A's elements.

Axiom 7 (well-ordering). Suppose *Ord* identifies ordinals and $B \leftrightarrow A$ is the bijective mappings from B to A. Assume $\forall\, A.\, \exists\, \alpha\, f.\, \mathit{Ord}(\alpha) \wedge f \in \alpha \leftrightarrow A$; i.e. every set can be well-ordered. $\square$

Because f is not unique, a well-ordering primitive could make λ_{ZFC}'s semantics nondeterministic. Fortunately, the existence of a cardinality operator is equivalent to well-ordering [19], so we will give λ_{ZFC} a cardinality primitive.

The **cardinality** of a set A is the smallest ordinal that can be put in bijection with A. Formally, if F contains A's well-orderings, $|A| = \bigcap\{\mathit{domain}(f) \mid f \in F\}$.

$$e \;::=\; n \mid v \mid e\ e \mid \mathsf{if}\ e\ e\ e \mid e \in e \mid \bigcup e \mid \mathsf{take}\ e \mid \mathcal{P}\ e \mid \mathsf{image}\ e\ e \mid \mathsf{card}\ e$$
$$v \;::=\; \mathsf{false} \mid \mathsf{true} \mid \lambda.\, e \mid \emptyset \mid \omega \qquad n \;::=\; 0 \mid 1 \mid 2 \mid \cdots$$

Fig. 1: The definition of $\lambda^{-}_{\mathrm{ZFC}}$, which represents countably many λ_{ZFC} terms.

2.3 Infinity's Infinity: An Inaccessible Cardinal

The set $\mathcal{V}(\omega)$ of hereditarily finite sets is closed under powerset, union, replacement (with predicates restricted to $\mathcal{V}(\omega)$), and cardinality. It is also **transitive**: if $A \in \mathcal{V}(\omega)$, then $x \in \mathcal{V}(\omega)$ for all $x \in A$. These closure properties make it a **Grothendieck universe**: a set that acts like a set of all sets.

λ_{ZFC}'s values should contain ω and be closed under its primitives. But a Grothendieck universe containing ω cannot be proved from the typical axioms. If it exists, it must be equal to $\mathcal{V}(\kappa)$ for some **inaccessible cardinal** κ.

Axiom 8 (inaccessible cardinal). Suppose $GU(V)$ if and only if V is a Grothendieck universe. Add 'κ' and assume $Ord(\kappa) \wedge (\kappa > \omega) \wedge GU(\mathcal{V}(\kappa))$. □

We call the sets in $\mathcal{V}(\kappa)$ **hereditarily accessible**.

Inaccessible cardinals are not usually assumed but are widely believed consistent. Set theorists regard them as no more dangerous than ω. Interpreting category theory with small and large categories, second-order set theory, or CIC in first-order set theory requires at least one inaccessible cardinal [20, 3, 21].

Constructing a set $A \notin \mathcal{V}(\kappa)$ requires assuming κ or an equivalent, so $\mathcal{V}(\kappa)$ easily contains most mathematics. In fact, most can be modeled well within $\mathcal{V}(2^{\omega})$; e.g. the model of $\mathbb{R}$ we define in Sect. 6 is in $\mathcal{V}(\omega + 11)$. Besides, if λ_{ZFC} needed to contain large cardinals, we could always assume even larger ones.

3 λ_{ZFC}'s Grammar

We define λ_{ZFC}'s terms in three steps. First, we define $\lambda^{-}_{\mathrm{ZFC}}$, a language of finite terms with primitives that correspond with the ZFC axioms. Second, we encode these terms as sets. Third, guided by the first two steps, we define λ_{ZFC} by defining its terms, most of which are infinite, as sets in $\mathcal{V}(\kappa)$.

Figure 1 shows $\lambda^{-}_{\mathrm{ZFC}}$'s grammar. Expressions e are typical: variables, values, application, if, and domain-specific primitives for membership, union, extraction (take), powerset, functional replacement (image), and cardinality. Values v are also typical: booleans and lambdas, and the domain-specific constants $\emptyset$ and ω.

In set theory, $\bigcup \{A\} = A$ holds for all A, so $\bigcup$ can extract the element from a singleton. In λ_{ZFC}, the encoding of $\bigcup \{A\}$ reduces to A only if A is an encoded set. Therefore, the primitives must include take, which extracts A from $\{A\}$. In particular, extracting a lambda from an ordered pair requires take.

We use De Bruijn indexes with 0 referring to the innermost binding. Because we will define λ_{ZFC} terms as well-founded sets, by Axiom 2, countably many indexes is sufficient for λ_{ZFC} as well as $\lambda^{-}_{\mathrm{ZFC}}$.

Distinct $t_{\text{var}}, t_{\text{app}}, t_{\text{if}}, t_{\in}, t_{\cup}, t_{\text{take}}, t_{\mathcal{P}}, t_{\text{image}}, t_{\text{card}}, t_{\text{set}}, t_{\text{atom}}, t_{\lambda}, t_{\text{false}}, t_{\text{true}}$

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\mathcal{F}[\![n]\!] & := & \langle t_{\text{var}}, n\rangle & \mathcal{F}[\![\emptyset]\!] & := & set(\varnothing) & \mathcal{F}[\![\omega]\!] & := & set(\omega) \\
\mathcal{F}[\![e_f\ e_x]\!] & := & \langle t_{\text{app}}, \mathcal{F}[\![e_f]\!], \mathcal{F}[\![e_x]\!]\rangle & \mathcal{F}[\![\mathsf{false}]\!] & := & a_{\text{false}} & a_{\text{false}} & := & \langle t_{\text{atom}}, t_{\text{false}}\rangle \\
\mathcal{F}[\![e_x \in e_A]\!] & := & \langle t_{\in}, \mathcal{F}[\![e_x]\!], \mathcal{F}[\![e_A]\!]\rangle & \mathcal{F}[\![\mathsf{true}]\!] & := & a_{\text{true}} & a_{\text{true}} & := & \langle t_{\text{atom}}, t_{\text{true}}\rangle \\
\cdots & & & set(A) & = & \langle t_{\text{set}}, \{set(x) \mid x \in A\}\rangle & & &
\end{array}$$

Fig. 2: The semantic function $\mathcal{F}[\![\cdot]\!]$ from λ^{-}_{ZFC} terms to λ_{ZFC} terms.

Figure 2 shows part of the meta-metalanguage function $\mathcal{F}[\![\cdot]\!]$ that encodes λ^{-}_{ZFC} terms as λ_{ZFC} terms. It distinguishes sorts of terms in the standard way, by pairing them with tags; e.g. if t_{set} is the "set" tag, then $\langle t_{\text{set}}, \varnothing\rangle$ encodes $\varnothing$.

To recursively tag sets, we add the axiom $set(A) = \langle t_{\text{set}}, \{set(x) \mid x \in A\}\rangle$. The **well-founded recursion theorem** proves that for all A, $set(A)$ exists, so this axiom is a conservative extension. The actual proof is tedious, but in short, *set* is structurally recursive. Now $set(\varnothing) = \langle t_{\text{set}}, \varnothing\rangle$ and $set(\omega)$ encodes ω.

3.1 An Infinite Set Rule For Finite BNF Grammars

There is no sensible reduction relation for λ^{-}_{ZFC}. (For example, $\mathcal{P}\ \emptyset$ cannot correctly reduce to a value because no value in λ^{-}_{ZFC} corresponds to $\{\varnothing\}$.) The easiest way to ensure a reduction relation exists for λ_{ZFC} is to include encodings of all the sets in $\mathcal{V}(\kappa)$ as values.

To define λ_{ZFC}'s terms, we first extend BNF with a set rule: $\{y^{*\alpha}\}$, where α is a cardinal. Roughly, it means sets comprised of no more than α terms from the language of y. Formally, it means $\mathcal{P}_{<}(Y, \alpha) := \{x \in \mathcal{P}(Y) \mid |x| < \alpha\}$, where Y is a subset of y's language generated while building a least fixpoint.

Example 3 (finite sets). The grammar $h ::= \{h^{*\omega}\}$ should represent all hereditarily finite sets, or $\mathcal{V}(\omega)$. Intuitively, the single rule for h should be equivalent to countably many rules $h ::= \{\} \mid \{h\} \mid \{h, h\} \mid \{h, h, h\} \mid \cdots$.

Its language is the least fixpoint of $F_h(H) := \mathcal{P}_{<}(H, \omega)$. Further on, we will prove that F_h's least fixpoint is $\mathcal{V}(\omega)$ using a general theorem. ♢

Example 4 (accessible sets). The language of $a ::= \{a^{*\kappa}\}$ is the least fixpoint of $F_a(A) := \mathcal{P}_{<}(A, \kappa)$, which should be $\mathcal{V}(\kappa)$. ♢

The following theorem schemas will make it easy to find least fixpoints.

Theorem 1. *Let F be a unary function. Define V by transfinite recursion:*

$$\begin{array}{rcl@{\qquad}rcl}
V(0) & = & \varnothing & V(\beta) & = & \bigcup_{\alpha\in\beta} V(\alpha),\ \textit{limit ordinal } \beta \\
V(s(\alpha)) & = & F(V(\alpha)) & & &
\end{array} \tag{6}$$

*Let γ be an ordinal. If F is monotone on $V(\gamma)$, V is monotone on γ, and $V(\gamma)$ is a fixpoint of F, then $V(\gamma)$ is also the **least** fixpoint of F.*

$$
\begin{array}{rcl}
e & ::= & n \mid v \mid \langle t_{\text{app}}, e, e\rangle \mid \langle t_{\text{if}}, e, e, e\rangle \mid \langle t_{\in}, e, e\rangle \mid \langle t_{\cup}, e\rangle \mid \langle t_{\text{take}}, e\rangle \mid \langle t_{\mathcal{P}}, e\rangle \mid \\
 & & \langle t_{\text{image}}, e, e\rangle \mid \langle t_{\text{card}}, e\rangle \mid \langle t_{\text{set}}, \{e^{*\kappa}\}\rangle \\
v & ::= & a_{\text{false}} \mid a_{\text{true}} \mid \langle t_{\lambda}, e\rangle \mid \langle t_{\text{set}}, \{v^{*\kappa}\}\rangle \qquad n ::= \langle t_{\text{var}}, 0\rangle \mid \langle t_{\text{var}}, 1\rangle \mid \cdots
\end{array}
$$

Fig. 3: λ_{ZFC}'s grammar. Here, $\{e^{*\kappa}\}$ means sets comprised of no more than κ terms from the language of e.

Proof. By induction: successor case by monotonicity; limit by property of $\bigcup$. □

All the Fs we define are monotone. In particular, the interpretations of $\{y^{*\alpha}\}$ rules are monotone because $\mathcal{P}$ is monotone. Further, all the Fs we define give rise to a monotone V. Grammar terminals "seed" every iteration with singleton sets, and $\{y^{*\alpha}\}$ rules seed every iteration with $\varnothing$.

From here on, we write F^{α} instead of $V(\alpha)$ to mean α iterations of F.

Theorem 2. *Suppose a grammar with $\{y^{*\alpha}\}$ rules and iterating function F. Then F's least fixpoint is F^{γ}, where γ is a regular cardinal not less than any α.*

Proof. Fixpoint by Aczel [2, Theorem 1.3.4]; least fixpoint by Theorem 1. □

Example 5 (finite sets). Because ω is regular, by Theorem 2, F_h's least fixpoint is F_h^{ω}. Further, $F_h(H) = \mathcal{P}(H)$ for all hereditarily finite H, and $\mathcal{V}(\omega)$ is closed under $\mathcal{P}$, so $F_h^{\omega} = \mathcal{V}(\omega)$, the set of all hereditarily finite sets. ♢

Example 6 (accessible sets). By a similar argument, F_a's least fixpoint is $F_a^{\kappa} = \mathcal{V}(\kappa)$, the set of all hereditarily accessible sets. ♢

Example 7 (encoded accessible sets). The language of $v ::= \langle t_{\text{set}}, \{v^{*\kappa}\}\rangle$ is comprised of the *encodings* of all the hereditarily accessible sets. ♢

3.2 The Grammar of Infinite, Encoded Terms

There are three main differences between λ_{ZFC}'s grammar in Fig. 3 and λ_{ZFC}^{-}'s grammar in Fig. 1. First, λ_{ZFC}'s grammar defines a language of terms that are already encoded as sets. Second, instead of the symbols $\emptyset$ and ω, it includes, as values, encoded sets of values. Most of these value terms are infinite, such as the encoding of ω. Third, it includes encoded sets of *expressions*.

The language of n is $N := \{\langle t_{\text{var}}, i\rangle \mid i \in \omega\}$. The rules for e and v are mutually recursive. Interpreted, but leaving out some of e's rules, they are

$$
\begin{aligned}
F_e(E, V) &:= N \cup V \cup \{\langle t_{\text{app}}, e_f, e_x\rangle \mid \langle e_f, e_x\rangle \in E \times E\} \cup \cdots \cup \\
&\qquad \{\langle t_{\text{set}}, e\rangle \mid e \in \mathcal{P}_{<}(E, \kappa)\} \qquad (7) \\
F_v(E, V) &:= \{a_{\text{false}}, a_{\text{true}}\} \cup \{\langle t_{\lambda}, e\rangle \mid e \in E\} \cup \{\langle t_{\text{set}}, v\rangle \mid v \in \mathcal{P}_{<}(V, \kappa)\}
\end{aligned}
$$

To use Theorem 2, we need to iterate a single function. Note that the language pair $\langle E, V\rangle = \langle \{e, ...\}, \{v, ...\}\rangle$ is isomorphic to the single set of tagged terms

$$\frac{}{v \Downarrow v}\ (\text{val}) \quad \frac{e_f \Downarrow \langle t_\lambda, e_y\rangle \quad e_x \Downarrow v_x \quad e_y[0\backslash v_x] \Downarrow v_y}{\langle t_{\text{app}}, e_f, e_x\rangle \Downarrow v_y}\ (\text{ap}) \quad \frac{e_c \Downarrow a_{\text{true}} \quad e_t \Downarrow v_t}{\langle t_{\text{if}}, e_c, e_t, e_f\rangle \Downarrow v_t} \quad \frac{e_c \Downarrow a_{\text{false}} \quad e_f \Downarrow v_f}{\langle t_{\text{if}}, e_c, e_t, e_f\rangle \Downarrow v_f}\ (\text{if})$$

(a) Standard call-by-value reduction rules

$$\frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A) \quad e_x \Downarrow v_x \quad v_x \in \mathit{snd}(v_A)}{\langle t_\in, e_x, e_A\rangle \Downarrow a_{\text{true}}} \quad \frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A) \quad e_x \Downarrow v_x \quad v_x \notin \mathit{snd}(v_A)}{\langle t_\in, e_x, e_A\rangle \Downarrow a_{\text{false}}}\ (\text{in})$$

$$\frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A) \quad \forall v_x \in \mathit{snd}(v_A).\, V_{\text{set}}(v_x)}{\langle t_\cup, e_A\rangle \Downarrow \widehat{\bigcup}(v_A)}\ (\text{union}) \qquad \frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A)}{\langle t_{\mathcal{P}}, e_A\rangle \Downarrow \widehat{\mathcal{P}}(v_A)}\ (\text{pow})$$

$$\frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A) \quad e_f \Downarrow \langle t_\lambda, e_y\rangle \quad \widehat{I}(\langle t_\lambda, e_y\rangle, v_A) \Downarrow v_y}{\langle t_{\text{image}}, e_f, e_A\rangle \Downarrow v_y}\ (\text{image}) \qquad \frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A)}{\langle t_{\text{card}}, e_A\rangle \Downarrow \widehat{C}(v_A)}\ (\text{card})$$

$$\frac{E_{\text{set}}(e_A) \quad \forall e_x \in \mathit{snd}(e_A).\, \exists v_x.\, e_x \Downarrow v_x}{e_A \Downarrow \langle t_{\text{set}}, \{v_x \mid e_x \in \mathit{snd}(e_A) \wedge e_x \Downarrow v_x\}\rangle}\ (\text{set}) \qquad \frac{e_A \Downarrow \langle t_{\text{set}}, \{v_x\}\rangle}{\langle t_{\text{take}}, e_A\rangle \Downarrow v_x}\ (\text{take})$$

(b) λ_{ZFC}-specific rules

Fig. 4: Reduction rules defining λ_{ZFC}'s big-step, call-by-value semantics.

$EV = \{\langle 0, e\rangle, ..., \langle 1, v\rangle, ...\}$. Binary **disjoint union**, denoted $E \sqcup V$, creates such sets. We define F_{ev} by $F_{ev}(E \sqcup V) = F_e(E, V) \sqcup F_v(E, V)$. By Theorem 2, its least fixpoint is F_{ev}^κ, so we define E and V by $E \sqcup V = F_{ev}^\kappa$.

To make well-founded substitution easy, we will use capturing substitution, which does not capture when used on closed terms. Let $\mathit{Cl}(e)$ indicate whether a term is closed—this is structurally recursive. Then $E' := \{e \in E \mid \mathit{Cl}(e)\}$ and $V' := \{v \in V \mid \mathit{Cl}(v)\}$ contain only closed terms. Lastly, we define $\lambda_{\text{ZFC}} := E'$.

4 λ_{ZFC}'s Big-Step Reduction Semantics

We distinguish sets from other expressions using E_{set} and V_{set}, which merely check tags. We also lift set constructors to operate on encoded sets. For example, for cardinality, $\widehat{C}(v_A) := \mathit{set}(|\mathit{snd}(v_A)|)$ extracts the tagged set from v_A, applies $|\cdot|$, and recursively tags the resulting cardinal number. The rest are

$$\begin{aligned} \widehat{\mathcal{P}}(v_A) &:= \langle t_{\text{set}}, \{\langle t_{\text{set}}, v_x\rangle \mid v_x \in \mathcal{P}(\mathit{snd}(v_A))\}\rangle \\ \widehat{\bigcup}(v_A) &:= \langle t_{\text{set}}, \bigcup\{\mathit{snd}(v_x) \mid v_x \in \mathit{snd}(v_A)\}\rangle \\ \widehat{I}(v_f, v_A) &:= \langle t_{\text{set}}, \{\langle t_{\text{app}}, v_f, v_x\rangle \mid v_x \in \mathit{snd}(v_A)\}\rangle \end{aligned} \tag{8}$$

All but $\widehat{I}$ return values. Sets returned by $\widehat{I}$ are intended to be reduced further.

We use $e[n\backslash v]$ for De Bruijn substitution. Because e and v are closed, it is easy to define it using simple structural recursion on terms; it is thus conservative.

Figure 4 shows the reduction rules that define the reduction relation '$\Downarrow$'. Figure 4a has standard call-by-value rules: values reduce to themselves, and applications reduce by substitution. Figure 4b has the λ_{ZFC}-specific rules. Most

simply use V_{set} to check tags before applying a lifted operator. The (image) rule replaces each value v_x in the set v_A with an application, generating a set expression, and the (set) rule reduces all the terms inside a set expression.

To define '$\Downarrow$' as a least fixpoint, we adapt Aczel's treatment [2]. We first define a bounding set for '$\Downarrow$' using closed terms, or $\mathcal{U} := E' \times V'$, so that $\Downarrow \subseteq \mathcal{U}$.

The rules in Fig. 4 can be used to define a predicate $D(R, \langle e, v\rangle)$. This predicate indicates whether some reduction rule, after replacing every '$\Downarrow$' in its premise with the approximation R, derives the conclusion $e \Downarrow v$.[1] Using D, we define a function that derives new conclusions from the known conclusions in R:

$$F_{\Downarrow}(R) \;:=\; \{c \in \mathcal{U} \mid D(R, c)\} \tag{9}$$

For example, $F_{\Downarrow}(\varnothing) = \{\langle v, v\rangle \mid v \in V\}$, by the (val) rule. $F_{\Downarrow}(F_{\Downarrow}(\varnothing))$ includes all pairs of non-value expressions and the values they reduce to in one derivation, as well as $\{\langle v, v\rangle \mid v \in V\}$. Generally, (val) ensures that iterating $F_{\Downarrow}$ is monotone.

For $F_{\Downarrow}$ itself to be non-monotone, for some $R \subseteq R' \subseteq \mathcal{U}$, there would have to be a conclusion $c \in F_{\Downarrow}(R)$ that is not in $F_{\Downarrow}(R')$. In other words, having more known conclusions could falsify a premise. None of the rules in Fig. 4 can do so.

Because $F_{\Downarrow}$ is monotone and iterating it is monotone, we can define $\Downarrow := F_{\Downarrow}^{\gamma}$ for some ordinal γ. If λ_{ZFC} had only finite terms, $\gamma = \omega$ iterations would reach a fixpoint. But a simple countable term shows why '$\Downarrow$' cannot be $F_{\Downarrow}^{\omega}$.

Example 8 (countably infinite term). If s is the successor function in λ_{ZFC}, the term $t := \langle t_{\text{set}}, \{0, \langle t_{\text{app}}, s, 0\rangle, \langle t_{\text{app}}, s, \langle t_{\text{app}}, s, 0\rangle\rangle, ...\}\rangle$ should reduce to $set(\omega)$. The (set) rule's premises require each of t's subterms to reduce—using at least $F_{\Downarrow}^{\omega}$ because each subterm requires a finite, unbounded number of (ap) derivations. Though $F_{\Downarrow}^{s(\omega)}$ reduces t, for larger terms, we must iterate $F_{\Downarrow}$ much further. ♢

Theorem 3. *$\Downarrow := F_{\Downarrow}^{\kappa}$ is the least fixpoint of $F_{\Downarrow}$.*

Proof. Fixpoint by Aczel [2, Theorem 1.3.4]; least fixpoint by Theorem 1. □

Lastly, ZFC theorems that do not depend on κ can be applied to λ_{ZFC} terms.

Theorem 4. *λ_{ZFC}'s set values and $\langle t_{\in}, \cdot, \cdot\rangle$ are a model of ZFC-κ.*

Proof. $\mathcal{V}(\kappa)$, a model of ZFC-κ, is isomorphic to $v ::= \langle t_{\text{set}}, \{v^{*\kappa}\}\rangle$. □

5 Syntactic Sugar and a Small Set Library

From here on, we write only λ_{ZFC}^{-} terms, assume $\mathcal{F}[\![\cdot]\!]$ is applied, and no longer distinguish λ_{ZFC}^{-} from λ_{ZFC}.

We use names instead of De Bruijn indexes and assume names are converted. We get alpha equivalence for free; for example, $\lambda \mathsf{x}.\,\mathsf{x} = \langle t_{\lambda}, \langle t_{\text{var}}, 0\rangle\rangle = \lambda \mathsf{y}.\,\mathsf{y}$.

λ_{ZFC} does not contain terms with free variables. To get around this technical limitation, we assume free variables are metalanguage names for closed terms.

[1] D is definable in first-order logic, but its definition does not aid understanding much.

We allow the primitives '$\in$', $\bigcup$, take, $\mathcal{P}$, image and card to be used as if they were functions. Enclosing infix operators in parenthesis refers to them as functions, as in $(\in)$. We partially apply infix functions using Haskell-like sectioning rules, so $(\mathsf{x} \in)$ means $\lambda \mathsf{A}.\, \mathsf{x} \in \mathsf{A}$ and $(\in \mathsf{A})$ means $\lambda \mathsf{x}.\, \mathsf{x} \in \mathsf{A}$.

We define first-order objects using ':=', as in $\mathsf{0} := \emptyset$, and syntax with ':$\equiv$', as in $\lambda\, x_1\, x_2 \ldots x_n.\, e :\equiv \lambda\, x_1.\, \lambda\, x_2 \ldots x_n.\, e$ to automatically curry. Function definitions expand to lambdas (using fixpoint combinators for recursion); for example, $\mathsf{x} = \mathsf{y} := \mathsf{x} \in \{\mathsf{y}\}$ and $(=) := \lambda \mathsf{x}\, \mathsf{y}.\, \mathsf{x} \in \{\mathsf{y}\}$ equivalently define $(=)$ in terms of $(\in)$. We destructure pairs implicitly in binding patterns, as in $\lambda\, \langle \mathsf{x}, \mathsf{y} \rangle.\, \mathsf{f}\ \mathsf{x}\ \mathsf{y}$.

To do anything useful, we need a small set library. The definitions are similar to the metalanguage definitions we omitted in Section 2, and we similarly elide most of the λ_{ZFC} definitions. However, some deserve special mention.

Because λ_{ZFC} has only *functional* replacement, we cannot define unbounded $\forall$ and $\exists$. But we can define bounded quantifiers in terms of bounded selection, or $\mathsf{select}\ \mathsf{f}\ \mathsf{A} := \bigcup\ (\mathsf{image}\ (\lambda \mathsf{x}.\, \mathsf{if}\ (\mathsf{f}\ \mathsf{x})\ \{\mathsf{x}\}\ \emptyset)\ \mathsf{A})$. We also define a bounded description operator $\iota\, x \in e_A.\, e_f :\equiv \mathsf{take}\ (\mathsf{select}\ (\lambda\, x.\, e_f)\ e_A)$. Note $\iota\, x \in e_A.\, e_f$ reduces only if $e_f \Downarrow \mathsf{true}$ for exactly one $\mathsf{x} \in e_A$.

Thus, converting a predicate to an object requires both unique existence and a bounding set. For example, if $\langle e_x, e_y \rangle :\equiv \{\{e_x\}, \{e_x, e_y\}\}$ defines ordered pairs, then $\mathsf{fst}\ \mathsf{p} := \iota \mathsf{x} \in (\bigcup\ \mathsf{p}).\, \exists \mathsf{y} \in (\bigcup\ \mathsf{p}).\, \mathsf{p} = \langle \mathsf{x}, \mathsf{y} \rangle$ takes the first element.

The ***set monad*** simulates nondeterministic choice. We define it by

$$\mathsf{return}_{\mathsf{set}}\ \mathsf{a}\ :=\ \{\mathsf{a}\} \qquad\qquad \mathsf{bind}_{\mathsf{set}}\ \mathsf{A}\ \mathsf{f}\ :=\ \bigcup\ (\mathsf{image}\ \mathsf{f}\ \mathsf{A}) \tag{10}$$

Using $\mathsf{bind}\ \mathsf{m}\ \mathsf{f} = \mathsf{join}\ (\mathsf{lift}\ \mathsf{f}\ \mathsf{m})$, evidently $\mathsf{lift}_{\mathsf{set}} := \mathsf{image}$ and $\mathsf{join}_{\mathsf{set}} := \bigcup$. The proofs of the monad laws follow the proofs for the list monad. We also define $\{x \in e_A\}.\, e_f :\equiv \mathsf{bind}_{\mathsf{set}}\ (\lambda\, x.\, e_f)\ e_A$, read "choose x in e_A, then e_f." For example, binary cartesian product is $\mathsf{A} \times \mathsf{B} := \{\mathsf{x} \in \mathsf{A}\}.\, \{\mathsf{y} \in \mathsf{B}\}.\, \mathsf{return}_{\mathsf{set}}\ \langle \mathsf{x}, \mathsf{y} \rangle$.

Every $\mathsf{f} \in \mathsf{A} \to \mathsf{B}$ is shaped $\mathsf{f} = \{\langle \mathsf{x}_1, \mathsf{y}_1 \rangle, \langle \mathsf{x}_2, \mathsf{y}_2 \rangle, \ldots\}$ and is total on A. To distinguish these hash tables from lambdas, we call them **mappings**. They can be applied by $\mathsf{ap}\ \mathsf{f}\ \mathsf{x} := \iota \mathsf{y} \in (\mathsf{range}\ \mathsf{f}).\, \langle \mathsf{x}, \mathsf{y} \rangle \in \mathsf{f}$, but we write just $\mathsf{f}\ \mathsf{x}$. We define $e_f|_{e_A} :\equiv \mathsf{image}\ (\lambda \mathsf{x}.\ \langle \mathsf{x}, e_f\ \mathsf{x} \rangle)\ e_A$ to convert a lambda or to restrict a mapping to e_A. We usually use $\lambda\, x \in e_A.\, e_y :\equiv (\lambda\, x.\, e_y)|_{e_A}$ to define mappings.

A sequence of A is a mapping $\mathsf{xs} \in \alpha \to \mathsf{A}$ for some ordinal α. For example, $\mathsf{ns} := \lambda \mathsf{n} \in \omega.\, \mathsf{n}$ is a countable sequence in $\omega \to \omega$ of increasing finite ordinals. We assume useful sequence functions like map, $\mathsf{map2}$ and drop are defined.

6 Example: The Reals From the Rationals

Here, we demonstrate that λ_{ZFC} is computationally powerful enough to construct the real numbers. For a clear, well-motivated, rigorous treatment in first-order set theory without lambdas, we recommend Abbott's excellent introductory text [1].

Assume we have a model $\mathbb{Q}, +_{\mathbb{Q}}, -_{\mathbb{Q}}, \times_{\mathbb{Q}}, \div_{\mathbb{Q}}$ of the rationals and rational arithmetic.[2] To get the reals, we close the rationals under countable limits.

[2] Though the λ_{ZFC} development of $\mathbb{Q}$ is short and elegant, it does not fit in this paper.

We represent limits of rationals with sequences in $\omega \to \mathbb{Q}$. To select only the converging ones, we must define what convergence means. We start with convergence to zero and equivalence. Given $\mathbb{Q}^+$, '$<_\mathbb{Q}$' and $|\cdot|_\mathbb{Q}$, define

$$\begin{aligned} \mathsf{conv\text{-}zero?_R\ xs} &:= \forall \varepsilon \in \mathbb{Q}^+.\, \exists\, \mathsf{N} \in \omega.\, \forall\, \mathsf{n} \in \omega.\, (\mathsf{N} \in \mathsf{n} \Rightarrow |\mathsf{xs\ n}|_\mathbb{Q} <_\mathbb{Q} \varepsilon) \\ \mathsf{xs} =_\mathsf{R} \mathsf{ys} &:= \mathsf{conv\text{-}zero?_R\ (map2\ (-_\mathbb{Q})\ xs\ ys)} \end{aligned} \tag{11}$$

So a sequence $\mathsf{xs} \in \omega \to \mathbb{Q}$ converges to zero if, for any positive ε, there is some index N after which all xs are smaller than ε. Two sequences are equivalent ($=_\mathsf{R}$) if their pointwise difference converges to zero.

We should be able to drop finitely many elements from a converging sequence without changing its limit. Therefore, a sequence of rationals converges to *something* when it is equivalent to all of its suffixes. We thus define an equivalent to the Cauchy convergence test, and use it to select the converging sequences:

$$\begin{aligned} \mathsf{conv?_R\ xs} &:= \forall\, \mathsf{n} \in \omega.\, \mathsf{xs} =_\mathsf{R} \mathsf{(drop\ n\ xs)} \\ \mathsf{R} &:= \mathsf{select\ conv?_R\ (\omega \to \mathbb{Q})} \end{aligned} \tag{12}$$

But R (equipped with the equivalence relation $=_\mathsf{R}$) is not the real numbers as they are normally defined: converging sequences in R may be equivalent but not equal. To decide real equality using λ_{ZFC}'s '=', we partition R into disjoint sets of equivalent sequences—we make a **quotient space**. Thus,

$$\begin{aligned} \mathsf{quotient\ A\ (=_A)} &:= \mathsf{image\ (\lambda x.\, select\ (=_A\ x)\ A)\ A} \\ \mathbb{R} &:= \mathsf{quotient\ R\ (=_R)} \end{aligned} \tag{13}$$

defines the reals with extensional equality.

To define real arithmetic, we must lift rational arithmetic to sequences and then to sets of sequences. The $\mathsf{map2}$ function lifts, say, $+_\mathbb{Q}$ to sequences, as in $(+_\mathsf{R}) := \mathsf{map2}\ (+_\mathbb{Q})$. To lift $+_\mathsf{R}$ to sets of sequences, note that sets of sequences are models of nondeterministic sequences, suggesting the set monad. We define $\mathsf{lift2_{set}\ f\ A\ B} := \{\mathsf{a} \in \mathsf{A}\}.\, \{\mathsf{b} \in \mathsf{B}\}.\, \mathsf{return_{set}\ (f\ a\ b)}$ to lift two-argument functions to the set monad. Now $(+) := \mathsf{lift2_{set}}\ (+_\mathsf{R})$, and similarly for the other operators.

Using $\mathsf{lift2_{set}}$ is atypical, so we prove that $\mathsf{A} + \mathsf{B} \in \mathbb{R}$ when $\mathsf{A} \in \mathbb{R}$ and $\mathsf{B} \in \mathbb{R}$, and similarly for the other operators. It follows from the fact that the rational operators lifted to sequences are surjective morphisms, and this theorem:

Theorem 5. *Suppose* $=_\mathsf{X}$ *is an equivalence relation on* X*, and define its quotient* $\mathbb{X} := \mathsf{quotient\ X\ (=_X)}$*. If* op *is surjective on* X *and a binary morphism for* $=_\mathsf{X}$*, then* $\mathsf{(lift2_{set}\ op\ A\ B)} \in \mathbb{X}$ *for all* $\mathsf{A} \in \mathbb{X}$ *and* $\mathsf{B} \in \mathbb{X}$*.*

Proof. Reduce to an equality. Case '$\subseteq$' by morphism; case '$\supseteq$' by surjection. □

Now for real limits. If $\mathbb{R}^+$, '$<$', and $|\cdot|$ are defined, we can define $\mathsf{conv\text{-}zero?_\mathbb{R}}$, which is like (11) but operates on real sequences $\mathsf{xs} \in \omega \to \mathbb{R}$. We then define $\mathsf{limit_\mathbb{R}\ xs} := \iota\, \mathsf{y} \in \mathbb{R}.\, \mathsf{conv\text{-}zero?_\mathbb{R}\ (map\ (-\ y)\ xs)}$ to calculate their limits.

From here, it is not difficult to treat $\mathbb{Q}$ and $\mathbb{R}$ uniformly by redefining $\mathbb{Q} \subset \mathbb{R}$.

7 Example: Computable Real Limits

Exact real computation has been around since Turing's seminal paper [18]. The novelty here is how we do it. We define the ***limit monad*** in λ_{ZFC} for expressing calculations involving limits, with $\mathsf{bind}_{\mathsf{lim}}$ defined in terms of a general limit. We then derive a limit-free, computable replacement $\mathsf{bind}'_{\mathsf{lim}}$. Replacing $\mathsf{bind}_{\mathsf{lim}}$ with $\mathsf{bind}'_{\mathsf{lim}}$ in a λ_{ZFC} term incurs proof obligations. If they can be met, the computable λ_{ZFC} term has the same limit as the original, uncomputable term.

In other words, entirely in λ_{ZFC}, we define uncomputable things, and gradually turn them into computable, directly implementable approximations.

The proof obligations are related to topological theorems [12] that we will import as lemmas. By Theorem 4, we are allowed to use them directly.

At this point, it is helpful to have a simple, informal type system, which we can easily add to the untyped λ_{ZFC}. $\mathsf{A} \Rightarrow \mathsf{B}$ is a lambda or mapping type. $\mathsf{A} \to \mathsf{B}$ is the set of total mappings from A to B. A set is a membership proposition.

7.1 The Limit Monad

We first need a universe $\mathbb{U}$ of values that is closed under sequencing; i.e. if $\mathsf{A} \subset \mathbb{U}$ then so is $\omega \to \mathsf{A}$. Define $\mathbb{U}$ as the language of $\mathsf{u} ::= \mathbb{R} \mid \omega \to \mathsf{u}$. A complete product metric $\delta : \mathbb{U} \Rightarrow \mathbb{U} \Rightarrow \mathbb{R}$ exists; therefore, a function $\mathsf{limit} : (\omega \to \mathbb{U}) \Rightarrow \mathbb{U}$ similar to $\mathsf{limit}_{\mathbb{R}}$ exists that calculates limits. These are all λ_{ZFC}-definable.

The limit monad's computations are of type $\omega \to \mathbb{U}$. The type does not imply convergence, which must be proved separately. Its run function is limit.

Example 9 (infinite series). Define $\mathsf{partial\text{-}sums} : (\omega \to \mathbb{R}) \Rightarrow (\omega \to \mathbb{R})$ first by $\mathsf{partial\text{-}sums'}\ \mathsf{xs} := \lambda\mathsf{n}.\ \mathsf{if}\ (\mathsf{n} = 0)\ (\mathsf{xs}\ 0)\ ((\mathsf{xs}\ \mathsf{n}) + (\mathsf{partial\text{-}sums'}\ \mathsf{xs}\ (\mathsf{n} - 1)))$. (The sequence is recursively defined, so we cannot use $\lambda\mathsf{n} \in \omega.\ e$ to immediately create it.) Then restrict its output: $\mathsf{partial\text{-}sums}\ \mathsf{xs} := (\mathsf{partial\text{-}sums'}\ \mathsf{xs})|_\omega$.

Now $\sum_{n\in\omega} e :\equiv \mathsf{limit}\ (\mathsf{partial\text{-}sums}\ \lambda n \in \omega.\ e)$, or the limit of partial sums. Even if xs converges, $\mathsf{partial\text{-}sums}\ \mathsf{xs}$ may not; e.g. if $\mathsf{xs} = \lambda\mathsf{n} \in \omega.\ \frac{1}{\mathsf{n}+1}$. ♢

The limit monad's $\mathsf{return}_{\mathsf{lim}} : \mathbb{U} \Rightarrow (\omega \to \mathbb{U})$ creates constant sequences, and its $\mathsf{bind}_{\mathsf{lim}} : (\omega \to \mathbb{U}) \Rightarrow (\mathbb{U} \Rightarrow (\omega \to \mathbb{U})) \Rightarrow (\omega \to \mathbb{U})$ simply takes a limit:

$$\mathsf{return}_{\mathsf{lim}}\ \mathsf{x} \ :=\ \lambda\mathsf{n} \in \omega.\mathsf{x} \qquad \mathsf{bind}_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{f} \ :=\ \mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}) \tag{14}$$

The left identity and associativity monad laws hold using '=' for equivalence. However, right identity holds only in the limit, so we define equivalence by $\mathsf{xs} =_{\mathsf{lim}} \mathsf{ys} \ :=\ \mathsf{limit}\ \mathsf{xs} = \mathsf{limit}\ \mathsf{ys}$.

Example 10 (lifting). Define $\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs} := \mathsf{bind}_{\mathsf{lim}}\ \mathsf{xs}\ \lambda\mathsf{x}.\ \mathsf{return}_{\mathsf{lim}}\ (\mathsf{f}\ \mathsf{x})$, as is typical. Substituting $\mathsf{bind}_{\mathsf{lim}}$ and reducing reveals that $\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}) = \mathsf{limit}\ (\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs})$. That is, using $\mathsf{lift}_{\mathsf{lim}}$ pulls limit out of f's argument. ♢

Example 11 (exponential). The Taylor series expansion of the exponential function is $\mathsf{exp\text{-}seq} : \mathbb{R} \Rightarrow (\omega \to \mathbb{R})$, defined by $\mathsf{exp\text{-}seq}\ \mathsf{x} := \mathsf{partial\text{-}sums}\ \lambda\mathsf{n} \in \omega.\ \frac{\mathsf{x}^\mathsf{n}}{\mathsf{n}!}$. It always converges, so $\mathsf{limit}\ (\mathsf{exp\text{-}seq}\ \mathsf{x}) = \sum_{\mathsf{n}\in\omega} \frac{\mathsf{x}^\mathsf{n}}{\mathsf{n}!} = \mathsf{exp}\ \mathsf{x}$ for $\mathsf{x} \in \mathbb{R}$. To exponentiate converging sequences, define $\mathsf{exp}_{\mathsf{lim}}\ \mathsf{xs} := \mathsf{bind}_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{exp\text{-}seq}$. ♢

7.2 The Computable Limit Monad

We derive the computable limit monad in two steps. In the first, longest step, we replace the limit monad's defining functions with those that do not use limit. But computations will still have type $\omega \to \mathbb{U}$, whose inhabitants are not directly implementable, so in the second step, we give them a lambda type.

We define $\mathsf{return}'_{\mathsf{lim}} := \mathsf{return}_{\mathsf{lim}}$. A drop-in, limit-free replacement for $\mathsf{bind}_{\mathsf{lim}}$ does not exist, but there is one that incurs three proof obligations. Without imposing rigid constraints on using $\mathsf{bind}_{\mathsf{lim}}$, we cannot meet them automatically. But we can separate them by factoring $\mathsf{bind}_{\mathsf{lim}}$ into $\mathsf{lift}_{\mathsf{lim}}$ and $\mathsf{join}_{\mathsf{lim}}$.

Limit-Free Lift. Substituting to get $\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs} = \mathsf{return}_{\mathsf{lim}}\ (\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}))$ exposes the use of limit. Removing it requires continuity and definedness.

Lemma 1 (continuity in metric spaces). *Let* $\mathsf{f} : \mathsf{A} \Rightarrow \mathsf{B}$ *with* A *a metric space. Then* f *is continuous at* $\mathsf{x} \in \mathsf{A}$ *if and only if for all* $\mathsf{xs} \in \omega \to \mathsf{A}$ *for which* $\mathsf{limit}\ \mathsf{xs} = \mathsf{x}$ *and* f *is defined on all elements of* xs, $\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}) = \mathsf{limit}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs})$.

So if $\mathsf{f} : \mathbb{U} \Rightarrow \mathbb{U}$ is continuous at $\mathsf{limit}\ \mathsf{xs}$, and f is defined on all xs, then

$$\begin{aligned} \mathsf{limit}\ (\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs}) &= \mathsf{limit}\ (\mathsf{return}_{\mathsf{lim}}\ (\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}))) \\ &= \mathsf{limit}\ (\mathsf{return}_{\mathsf{lim}}\ (\mathsf{limit}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs}))) \\ &= \mathsf{limit}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs}) \end{aligned} \tag{15}$$

Thus, $\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs} =_{\mathsf{lim}} \mathsf{map}\ \mathsf{f}\ \mathsf{xs}$, so $\mathsf{lift}'_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs} := \mathsf{map}\ \mathsf{f}\ \mathsf{xs}$. Using $\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs}$ instead of $\mathsf{lift}'_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs}$ requires f to be continuous at $\mathsf{limit}\ \mathsf{xs}$ and defined on all xs.

Limit-Free Join. Using $\mathsf{join}\ \mathsf{m} = \mathsf{bind}\ \mathsf{m}\ \lambda \mathsf{x}.\, \mathsf{x}$ results in $\mathsf{join}_{\mathsf{lim}} = \mathsf{limit}$. Removing limit might seem hopeless—until we distribute it pointwise over xss.

Lemma 2 (limits of sequences). *Let* $\mathsf{f} \in \omega \to \omega \to \mathsf{A}$, *where* $\omega \to \mathsf{A}$ *has a product topology. Then* $\mathsf{limit}\ \mathsf{f} = \lambda \mathsf{n} \in \omega.\, \mathsf{limit}\ (\mathsf{flip}\ \mathsf{f}\ \mathsf{n})$, *where* $\mathsf{flip}\ \mathsf{f}\ \mathsf{x}\ \mathsf{y} := \mathsf{f}\ \mathsf{y}\ \mathsf{x}$.

A countable product metric defines a product topology, so $\mathsf{join}_{\mathsf{lim}}\ \mathsf{xss} := \lambda \mathsf{n} \in \omega.\, \mathsf{limit}\ (\mathsf{flip}\ \mathsf{xss}\ \mathsf{n})$. Now we can remove limit by restricting $\mathsf{join}_{\mathsf{lim}}$'s input.

Definition 1 (uniform convergence). *A sequence* $\mathsf{f} \in \omega \to \omega \to \mathbb{U}$ *converges* ***uniformly*** *if* $\forall\, \varepsilon \in \mathbb{R}^+.\, \exists\, \mathsf{N} \in \omega.\, \forall\, \mathsf{n}, \mathsf{m} > \mathsf{N}.\, (\delta\ (\mathsf{f}\ \mathsf{n}\ \mathsf{m})\ (\mathsf{limit}\ (\mathsf{f}\ \mathsf{n}))) < \varepsilon$.

Lemma 3 (collapsing limits). *If* $\mathsf{f} \in \omega \to \omega \to \mathbb{U}$ *converges uniformly, and* $\mathsf{r}, \mathsf{s} : \omega \Rightarrow \omega$ *increase, then* $\mathsf{limit}\ \lambda \mathsf{n} \in \omega.\, \mathsf{limit}\ (\mathsf{f}\ \mathsf{n}) = \mathsf{limit}\ \lambda \mathsf{n} \in \omega.\, \mathsf{f}\ (\mathsf{r}\ \mathsf{n})\ (\mathsf{s}\ \mathsf{n})$.

So if $\mathsf{flip}\ \mathsf{xss}$ converges uniformly, then

$$\begin{aligned} \mathsf{limit}\ (\mathsf{join}_{\mathsf{lim}}\ \mathsf{xss}) &= \mathsf{limit}\ \lambda \mathsf{n} \in \omega.\, \mathsf{limit}\ (\mathsf{flip}\ \mathsf{xss}\ \mathsf{n}) \\ &= \mathsf{limit}\ \lambda \mathsf{n} \in \omega.\, \mathsf{flip}\ \mathsf{xss}\ (\mathsf{r}\ \mathsf{n})\ (\mathsf{s}\ \mathsf{n}) \end{aligned} \tag{16}$$

We define $\mathsf{join}'_{\mathsf{lim}} : (\omega \to \omega \to \mathbb{U}) \Rightarrow (\omega \to \mathbb{U})$ by $\mathsf{join}'_{\mathsf{lim}}\ \mathsf{xss} := \lambda \mathsf{n} \in \omega.\, \mathsf{xss}\ \mathsf{n}\ \mathsf{n}$. Replacing $\mathsf{join}_{\mathsf{lim}}\ \mathsf{xss}$ with $\mathsf{join}'_{\mathsf{lim}}\ \mathsf{xss}$ requires that $\mathsf{flip}\ \mathsf{xss}$ converge uniformly.

Limit-Free Bind. Define $\mathsf{bind}'_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{f} := \mathsf{join}'_{\mathsf{lim}}\ (\mathsf{lift}'_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs})$. It inherits obligations to prove that f is continuous at $\mathsf{limit}\ \mathsf{xs}$ and defined on all xs, and to prove that $\mathsf{flip}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs})$ converges uniformly.

Example 12 (exponential cont.). Define $\mathsf{exp}'_{\mathsf{lim}}$ by replacing $\mathsf{bind}_{\mathsf{lim}}$ by $\mathsf{bind}'_{\mathsf{lim}}$ in $\mathsf{exp}_{\mathsf{lim}}$, so $\mathsf{exp}'_{\mathsf{lim}}\ \mathsf{xs} := \mathsf{bind}'_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{exp\text{-}seq}$. We now meet the proof obligations.

Lemma 4. *Let* $\mathsf{f} : \mathsf{A} \Rightarrow (\omega \to \mathsf{B})$. *If* $\omega \to \mathsf{B}$ *has a product topology, then* f *is continuous if and only if* $(\mathsf{flip}\ \mathsf{f})\ \mathsf{n}$ *is continuous for every* $\mathsf{n} \in \omega$.

We have a product topology, so for the first obligation, pointwise continuity is enough. Let $\mathsf{g} := \mathsf{flip}\ \mathsf{exp\text{-}seq}$. Every $\mathsf{g}\ \mathsf{n}$ is a finite polynomial, and thus continuous. The second obligation, that $\mathsf{exp\text{-}seq}$ is defined on all xs, is obvious. The third, that $\mathsf{flip}\ (\mathsf{map}\ \mathsf{exp\text{-}seq}\ \mathsf{xs})$ converges uniformly, can be proved using the Weierstrass M test [1, Theorem 6.4.5]. ♢

Example 13 (π). The definition of $\mathsf{arctan}_{\mathsf{lim}}$ is like $\mathsf{exp}_{\mathsf{lim}}$'s. Defining $\mathsf{arctan}'_{\mathsf{lim}}$, including proving correctness, is like defining $\mathsf{exp}'_{\mathsf{lim}}$. To compute π, we use

$$\begin{aligned}\pi_{\mathsf{lim}} \ :=\ & ((\mathsf{return}_{\mathsf{lim}}\ 16) \times_{\mathsf{lim}} (\mathsf{arctan}_{\mathsf{lim}}\ (\mathsf{return}_{\mathsf{lim}}\ \tfrac{1}{5}))) -_{\mathsf{lim}} \\ & ((\mathsf{return}_{\mathsf{lim}}\ 4) \times_{\mathsf{lim}} (\mathsf{arctan}_{\mathsf{lim}}\ (\mathsf{return}_{\mathsf{lim}}\ \tfrac{1}{239})))\end{aligned} \tag{17}$$

where $(\cdot)_{\mathsf{lim}}$ are lifted arithmetic operators. Because (17) does not directly use $\mathsf{bind}_{\mathsf{lim}}$, defining the limit-free π'_{lim} imposes no proof obligations. ♢

In general, using functions defined in terms of $\mathsf{bind}'_{\mathsf{lim}}$ requires little more work than using functions on finite values. The implicit limits are pulled outward and collapse on their own, hidden within monadic computations.

Computable Sequences. Lambdas are the simplest model of $\omega \to \mathbb{U}$. After manipulating some terms, we define the final, computable limit monad by $\mathsf{return}'_{\mathsf{lim}}\ \mathsf{x} := \lambda \mathsf{n}.\, \mathsf{x}$ and $\mathsf{bind}'_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{f} := \lambda \mathsf{n}.\, \mathsf{f}\ (\mathsf{xs}\ \mathsf{n})\ \mathsf{n}$. Computations have type $\omega \Rightarrow \mathbb{U}'$, where $\mathbb{U}'$ contains countable sequences of rationals.

Implementation. We have transliterated $\mathsf{return}'_{\mathsf{lim}}$, $\mathsf{bind}'_{\mathsf{lim}}$, $\mathsf{exp}'_{\mathsf{lim}}$, $\mathsf{arctan}'_{\mathsf{lim}}$ and π'_{lim} into Racket [7], using its built-in models of ω and $\mathbb{Q}$. Even without optimizations, $\pi'_{\mathsf{lim}}\ 141$ yields a rational approximation in a few milliseconds that is correct to 200 digits. More importantly, $\mathsf{exp}'_{\mathsf{lim}}$, $\mathsf{arctan}'_{\mathsf{lim}}$ and π'_{lim} are almost identical to their counterparts in the uncomputable limit monad, and meet their proof obligations. The code is clean, short, correct and reasonably fast, and resides in a directory named `flops2012` at `https://github.com/ntoronto/plt-stuff/`.

8 Related Work

O'Connor's completion monad [13] is quite similar to the limit monad. Both operate on general metric spaces and compute to arbitrary precision. O'Connor

starts with computable approximations and completes them using a monad. Implementing it in Coq took five months. It is certainly correct.

We start with a monad for exact values and define a computable replacement. It was two weeks from conception to implementation. Between directly using well-known theorems, and deriving the computable monad from the uncomputable monad without switching languages, we are as certain as we can be without mechanically verifying it. We have found our middle ground.

Higher-order logics such as HOL [11], CIC [5], MT [4] (Map Theory) and EFL* [6] continue Church's programme to found mathematics on the lambda calculus. Like λ_{ZFC}, interpreting them in set theory seems to require a slightly stronger theory than plain ZFC. HOL and CIC ensure consistency using types, and use the Curry-Howard correspondence to extract programs.

MT and EFL* are more like λ_{ZFC} in that they are untyped. MT ensures consistency partly by making nontermination a truth value, and EFL* partly by tagging propositions. Both support classical reasoning. MT and EFL* are interpreted in set theory using a straightforward extension of Scott-style denotational semantics to κ-sized domains, while λ_{ZFC} is interpreted in set theory using a straightforward extension of operational semantics to κ-sized relations.

The key difference between λ_{ZFC} and these higher-order logics is that λ_{ZFC} is not a logic. It is a programming language with infinite terms, which by design includes a transitive model of set theory (Theorem 4). Therefore, ZFC theorems can be applied to its set-valued terms with only trivial interpretation, whereas the interpretation it takes to apply ZFC theorems to lambda terms that represent sets in MT or EFL* can be highly nontrivial. Applying a ZFC theorem in HOL or CIC requires re-proving it to the satisfaction of a type checker.

The infinitary lambda calculus [10] has "infinitely deep" terms. Although it exists for investigating laziness, cyclic data, and undefinedness in finitary languages, it is possible to encode uncomputable mathematics in it. In λ_{ZFC}, such up-front encodings are unnecessary.

Hypercomputation [14] describes many Turing machine extensions, including completion of transfinite computations. Much of the research is for discovering the properties of computation in physically plausible extensions. While λ_{ZFC} might offer a civilized way to program such machines, we do not think of our work as hypercomputation, but as approaching computability from above.

9 Conclusions and Future Work

We defined λ_{ZFC}, which can express essentially anything constructible in contemporary mathematics, in a way that makes it compatible with existing first-order theorems. We demonstrated that it makes deriving computational meaning easier by defining the limit monad in it, deriving a computable replacement, and computing real numbers to arbitrary accuracy with acceptable speed.

Our main future work is using λ_{ZFC} to define languages for Bayesian inference, then deriving implementations that compute converging probabilities. We

have already done this successfully for countable Bayesian models [17]. Having built our previous work's foundation, we can now proceed with it.

Overall, we no longer have to hold back when a set-theoretic construction could be defined elegantly with untyped lambdas or recursion, or generalized precisely with higher-order functions. If we can derive a computable replacement, we might help someone in Cantor's Paradise compute the apparently uncomputable.

References

1. Abbott, S.: Understanding Analysis. Springer (2001)
2. Aczel, P.: An introduction to inductive definitions. Studies in Logic and the Foundations of Mathematics 90, 739–782 (1977)
3. Barras, B.: Sets in Coq, Coq in sets. Journal of Formalized Reasoning 3(1) (2010)
4. Berline, C., Grue, K.: A κ-denotational semantics for Map Theory in ZFC+SI. Theoretical Computer Science 179(1–2), 137–202 (1997)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, Springer Verlag (2004), http://www.labri.fr/publications/l3a/2004/BC04
6. Flagg, R.C., Myhill, J.: A type-free system extending ZFC. Annals of Pure and Applied Logic 43, 79–97 (1989)
7. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. (2010), http://racket-lang.org/tr1/
8. Hrbacek, K., Jech, T.: Introduction to set theory. Pure and Applied Mathematics, M. Dekker (1999)
9. Hurd, J.: Formal Verification of Probabilistic Algorithms. Ph.D. thesis, University of Cambridge (2002)
10. Kennaway, R., Klop, J.W., Sleep, M.R., jan De Vries, F.: Infinitary lambda calculus. Theoretical Computer Science 175, 93–125 (1997)
11. Leivant, D.: Higher order logic. In: In Handbook of Logic in Artificial Intelligence and Logic Programming. pp. 229–321. Clarendon Press (1994)
12. Munkres, J.R.: Topology. Prentice Hall, second edn. (2000)
13. O'Connor, R.: Certified exact transcendental real number computation in Coq. In: TPHOLs'08. pp. 246–261 (2008)
14. Ord, T.: The many forms of hypercomputation. Applied Mathematics and Computation 178, 143–153 (2006)
15. Paulson, L.C.: Set theory for verification: II. Induction and recursion. Journal of Automated Reasoning 15, 167–215 (1995)
16. Paulson, L.C.: Set theory for verification: I. From foundations to functions. Journal of Automated Reasoning 11, 353–389 (1993)
17. Toronto, N., McCarthy, J.: From Bayesian notation to pure Racket, via measure-theoretic probability in λ_{ZFC}. In: Implementation and Application of Functional Languages (2010)
18. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. In: Proceedings of the London Mathematical Society. vol. 42, pp. 230–265 (1936)
19. Tzouvaras, A.: Cardinality without enumeration. Studia Logica: An International Journal for Symbolic Logic 80(1), 121–141 (June 2005)
20. Uzquiano, G.: Models of second-order Zermelo set theory. The Bulletin of Symbolic Logic 5(3), 289–302 (1999)
21. Werner, B.: Sets in types, types in sets. In: TACS'97. pp. 530–546 (1997)