An Integration of Axiomatic Set Theory with ACL2

Matt Kaufmann

UT Austin (retired)

April 11, 18, and 25, 2025

Outline

Outline

Outline

General Information

Thanks to Eric Smith and Kestrel for hosting and recording.

- This could kick off an online seminar series....

About this talk:

- Talk info is on the seminar page. We'll go there from the ACL2 home page and review the abstract.
- Please ask questions (with voice, not Zoom chat). **NOTE**: I am trying not to assume any background in ZF set theory.
- This is **work in progress** (e.g., no comparison with Isabelle/ZF or others).

Collaborators are welcome! I'll mention potential future work.

For more info see :DOC zfc, :DOC zfc-model, and the books: `books/projects/set-theory/`.

Books use no trust tags and required **no ACL2 changes**.

Motivation

Zermelo Fraenkel (ZF) set theory is an established, intuitive foundation for mathematics.

Personal motivation: Combines my logic background (40 to 50 years ago!) with my current focus, ACL2.

- I've always been a bit bothered by the built-in ground-zero theory – ACL2 isn't a *pure* first-order prover.

Key new insight last Fall: ACL2 can be a pure set-theory prover by encoding ACL2 primitives and data into set theory.

Additional motivation: Provides a vehicle for embedding higher-order logic (HOL) developments into ACL2.

- That could be the subject of future talks.

About Set Theory and ACL2

ACL2 objects are represented as sets.

- Natural numbers are Zermelo (von Neumann) ordinals:
  0 is the empty set, $\{\}$;
  $1 = \{0\}$;
  $2 = \{0, 1\}$;
  and in general
  $n = \{0, 1, ..., n-1\}$.
- Other ACL2 objects are encoded as discussed later, e.g.:
  - `Cons` is represented using the Kuratowski ordered pair:
    `(cons` x y`)` $= \{\{x\}, \{x, y\}\}$
  - $-3 = \{0, 1, (3\ .\ 0)\}$
- There are infinite objects but we can't compute with them, or with set membership, etc.

Let's look at this picture from Wikipedia:
$V_{\omega * \omega}$

Examples

Here we touch on two examples.
Note that these are in the `"ZF"` package.

- Classical set theory example: Cantor's theorem (Let's look briefly at the certifiable book, `cantor.lisp`); we'll revisit it later after providing more background.
- "Higher-order function" example: `map`
  - We'll look at `(defun map ...)` in `base.lisp` and the two theorems following it. First note:
    - In `(map f lst)`, think of `f` as a set of ordered pairs and `lst` as an ACL2 list.
    - I'll explain later how `(defthmz ...  :props ...)` can be viewed as `(defthm ...)`.
  - We'll look at `zify.lisp` to see an application of `map` to the Fibonacci function.
  - Later we may look at `foldr.lisp`.

Outline

Outline

ZFG

Goal: Provide a platform for efficient set-theory reasoning.

- The axioms need justification, but need not be minimal.
  - Example: The Axiom of Infinity of ZF says that there is a set containing the empty set and closed under the operation $n \mapsto n \cup \{n\}$, but we axiomatize ω to be a specific such set.

ZFG is ZF plus a *global choice* axiom.

Let's look at the exports in the first `encapsulate` form in `base.lisp`, up to "Embedding of ACL2 data types".

- Notice the local witness of `nil` for `zfc`, which serves as a hypothesis!
- A metatheoretic argument provides a meaningful interpretation for which `(zfc)` is true.
- Not included there: Comprehension (Subset) or Replacement (equivalently, Collection) schemes of ZF (to be discussed later)

OUTLINE

Outline

Review of First Talk

This work supports ACL2 as a logic and prover for set theory, ZFG (Zermelo-Fraenkel with global choice).

In ZFG we *define* the ACL2 numbers, characters, strings, and symbols (the *good atoms*):

- Naturals are finite ordinals $n = \{0, ..., n-1\}$;
- `Cons` is represented using the Kuratowski ordered pair, `(cons` x y`)` $= \{\{x\}, \{x, y\}\}$;
- $-3 = \{0, 1, (3\ .\ 0)\}$;
- etc.

For more info see: last week's slides and talk (links are on the ACL2 seminar page), :DOC zfc, :DOC zfc-model, and the books in `books/projects/set-theory/`.

We'll look again at the initial `encapsulate` event in `base.lisp`.

It introduces *hypothesis function* `zfc` and primitives `in`, `pair`, `min-in`, `union`, `omega`, and `powerset`, along with `subset` and some basic axioms.

Outline

Outline

Logical Overview

For this work, I view the logical foundation of ACL2 as first-order set theory, specifically, ZFG.

ZFG is powerful: All built-in ACL2 constants and functions, e.g. including `natp`, `expt`, `consp`, `cons`, `symbolp`, etc. can be *defined* in ZFG.

If time and interest permit, I might lay out rigorous details. The next slide makes a start.

I'll refer to these foundations — where ACL2 objects are encoded as sets and ACL2 functions are defined in ZFG — as *our underlying set theory*.

Encoding ACL2 Objects in Set Theory

Let's look at the rest of that initial `encapsulate` in `base.lisp` to see how ACL2 data type recognizers are defined — and also at the definitions of `relation-p`, `funp`, and `apply` after that.

TIP: Note, as in `funp`, the use of `non-exec` in `defun-sk` to support `guard` verification.

Outline

Outline

Comprehension in ZF

The *Comprehension* (or *Subset*) scheme of ZF says that the intersection of a predicate with a set is a set.

- Informally: $\{a \in x : P(a)\}$ is a set.
- Formal statement, for each formula P with y not free: $\forall x \exists y \forall a (a \in y \Leftrightarrow (a \in x \wedge P))$
- I'll call x the *bounding set*.

ZSUB EXAMPLE (1)

From `base.lisp`:

```
; The following defines the Cartesian product
; (prod2 a b)
; as:
; {p \in (powerset (powerset (union2 a b))) :
;   (prod-member p a b)}

(zsub prod2 (a b)
      p
      (powerset (powerset (union2 a b)))
      (prod-member p a b)
      )
```

For Comprehension (and `zsub`), we always need a *bounding set*. Why does `(powerset (powerset (union2 a b)))` serve that purpose?

ZSUB EXAMPLE (2)

Again, why is `a` $\times$ `b` contained in `(powerset (powerset (union2 a b)))`?

Consider the ordered pair $\{\{x\},\{x,y\}\} \in$ `a` $\times$ `b`, where $x \in$ `a` and $y \in$ `b`.

Both $\{x\}$ and $\{x,y\}$ are subsets of `(union2 a b)`, hence it's in `(powerset (union2 a b))`.

So $\{\{x\},\{x,y\}\}$ is a subset of `(powerset (union2 a b))`, hence it's in `(powerset (powerset (union2 a b)))`.

ZSUB EXAMPLE (3)

```
(zsub prod2 (a b)
      p
      (powerset (powerset (union2 a b)))
      (prod-member p a b)
      )
```

Let's see how this call of `zsub` expands, using `:trans1` and focusing on `PROD2$COMPREHENSION`.

More zsub Examples

As time permits we'll take a quick look at more examples in `base.lisp`:

`domain`, `inverse`, `image`, `compose`

ZFC-TABLE

Recall that the `prod2` example above generates:

```
(TABLE ZFC-TABLE
  'PROD2$PROP
  '(ZSUB PROD2 (A B)
         P
         (POWERSET (POWERSET (UNION2 A B)))
         (PROD-MEMBER P A B)))
```

Key property: Every key of `zfc-table` is a zero-ary function symbol that returns true in our underlying set theory.

Thus: The `table` guard of `zfc-table` checks that `prod2$prop` can be assumed to hold by the Comprehension scheme.

DEFTHMZ AND :PROPS

`Defthmz` (here, "z" to suggest "ZF") is just `defthm` except for an extra `:props` argument.

- The value of `:props` must be a list of keys of `zfc-table`.
- In our underlying set theory, all `:props` functions are true — we can ignore them!
  - After all, adding a bunch of `T` hypotheses has no logical effect.
- The default value for `:props` is `(zfc)`.
- `Defthmdz` and `thmz` similarly extend `defthmd` and `thm` (respectively) with a `:props` argument.

DEFTHMZ EXAMPLES

Use `:trans1` to look at examples in `base.lisp`, e.g., `ordinal-p-omega` and `in-prod2`.

`Make-event` tips from
`:trans1 (CHECK-PROPS DEFTHMZ (ZFC PROD2$PROP))`:

- **TIP**: Use `:expansion?` to avoid bloat in `.cert` file.
- **TIP**: Use `:on-behalf-of :quiet` to suppress noisy output
- **TIP**: Use `:check-expansion t` to ensure that the check is made even at `include-book` time.

SIMPLIFYING EXPORTS FROM ZSUB

Evaluate `:pe prod2$comprehension` and compare to `in-prod2`.

Let's look at the proof of `in-prod2` (file `base.lisp`), which simplifies `prod2$comprehension`.

Outline

Outline

ORDINALS

Let's look at these key events in the section "Omega is an ordinal" in `base.lisp` (with "`:guard t`" omitted).

```
(defun-sk in-is-linear (s)
  (forall (x y) (implies (and (in x s)
                              (in y s)
                              (not (equal x y)))
                         (or (in x y)
                             (in y x)))))
(defun-sk transitive (x)
  (forall a (implies (in a x)
                     (subset a x)))
  :rewrite :direct)
(defun ordinal-p (x)
  (and (in-is-linear x)
       (transitive x)))
(defthmz ordinal-p-omega (ordinal-p (omega)))
```

ORDINALS (CONTINUED)

See `ordinals.lisp` for a few more theorems about ordinals.
A key result:

```
(defthmz ordinal-trichotomy
  (implies (and (ordinal-p a)
                (ordinal-p b)
                (not (in a b))
                (not (in b a)))
           (equal (equal a b)
                  t))
  :props (zfc diff$prop)
  :hints ...)
```

ITERATED COMPOSITION

See `iterate.lisp`.

De Morgan's Laws Etc.

See `set-algebra.lisp`, e.g., De Morgan's Laws.

Function Spaces

See `fun-space.lisp`.

Reasoning about Free Variables and Quantifiers

Example: see `demo1.lsp` for a proof of the following theorem from `set-algebra.lisp`.

```
(defthmz domain-union2
    (equal (domain (union2 r s))
           (union2 (domain r) (domain s)))
    :props (zfc domain$prop)
    :hints ...)
```

Free Variables and Quantifiers (continued)

- **TIP**: Enable `extensionality-rewrite` to prove two sets are equal.
- **TIP**: Use `:otf-flg t` to see all checkpoints.
- **TIP**: Use `skip-proofs` to formulate lemmas (maybe with numbering scheme X-i-j-k...) and then see if those suffice.
- **TIP**: When proving a call of `subset`, open up that call by enabling `subset` or expanding that call. This strategy applies in general for `defun-sk` using `forall`. Also see :DOC quantifier-tutorial.
- **TIP**: Let forcing help you to find missing `:props`.
- **TIP**: Use proof-builder commands to explore, including `generalize`, `bash`, `lisp`, and `sr` (`show-rewrites`); see also :DOC proof-builder-commands-short-list.
- **TIP**: Use `:pl term` (much like using `sr` in the proof-builder).
- **TIP**: Accommodate proof-builder rewrites involving free variables by using `:restrict` hints.

Transfinite Induction

Time permitting, we'll talk about *epsilon-induction* and look at the macro `prove-inductive-suffices` and the examples below it in `induction.lisp`.

Transfinite induction on the ordinals is a special case of epsilon-induction.

Outline

Outline

Replacement Application #1: The Cumulative Hierarchy

The book `base.lisp` defines $V = \{\langle x, y\rangle : x \in \omega \wedge y = V_{map}(x)\}$ and then $V_\omega = \bigcup\{y : \{\langle x, y\rangle \in V\}$. [picture]

```
(defun v-map (n) ; uses ordinary ACL2 recursion!
  (declare (type (integer 0 *) n))
  (if (zp n)
      0
    (powerset (v-map (1- n)))))
```

Let's take a look using :trans1:

```
(zfn v ()                              ; name, args
     x y                               ; x, y
     (omega)                           ; bound for x
     (equal (equal y (v-map x)) ; relation on x, y
            t))
```

```
(defun v-omega () ;union of v-map(0), v-map(1), ...
  (declare (xargs :guard t))
  (union (image (v))))
```

Replacement in ZF and in ACL2

The Replacement Scheme of ZF says that a definable mapping F of a set A produces a set.

Here's what Wikipedia says, but we'll look at the picture.

$$\forall w_1, \ldots, w_n \, \forall A \, ([\forall x \in A \, \exists! y \, \phi(x, y, w_1, \ldots, w_n, A)] \implies \exists B \, \forall y \, [y \in B \Leftrightarrow \exists x \in A \, \phi(x, y, w_1, \ldots, w_n, A)])$$

ACL2 modifies the Replacement Scheme as follows. (This ACL2 version follows easily from the ZF axioms.)

- F can associate more than one value with the same input;
- the mapping can be **undefined** on any elements of A; and
- the result is a **set-theoretic function** — a set of ordered pairs — based on the restriction of F to A.

All Good ACL2 Objects Are in V_ω

Recognizer for "good ACL2 object":

```
(defun acl2p (x)
  (declare (xargs :guard t))
  (cond ((consp x) (and (acl2p (car x))
                        (acl2p (cdr x))))
        (t (not (bad-atom x)))))
```

Good ACL2 objects sit in V_ω (see `base.lisp`), which is critical for the use of `zsub` on the next slide:

```
(defthmz v-omega-contains-acl2p
  (implies (acl2p x)
           (in x (v-omega)))
  :props ... :hints ...)
```

The Good ACL2 Objects Form a Set

We'll use the following notion later (see `mirror` example).

$(\texttt{acl2}) = \{x \in V_\omega : \texttt{acl2p}(x)\}$ — see `zify.lisp`.

```
(zsub acl2 ()       ; name, args
      x             ; the variable
      (v-omega)     ; the bounding set
      (acl2p x))    ; the property

(extend-zfc-table ; Use :trans1 to see expansion.
 zify-prop
 prod2$prop domain$prop inverse$prop zfc)

(defthmz acl2p-is-acl2
; strengthens acl2$comprehension
  (equal (in x (acl2))
         (acl2p x))
  :props (zify-prop acl2$prop v$prop))

(in-theory (disable acl2$comprehension))
```

Replacement Example #2: Transitive Closure

We'll mostly skip this slide unless there is time for it.

A set is *transitive* if every member of a member is a member, i.e., every member is a subset.

```
(defun-sk transitive (x)
  (declare (xargs :guard t))
  (forall a (implies (in a x)
                     (subset a x)))
  :rewrite :direct)
```

File `tc.lisp` defines the *transitive closure* of a set s to be the least transitive set containing s.

Time permitting, we'll look at theorems labeled "A key theorem" in `tc.lisp`.

Perhaps we'll also look at the definition of `tc` in file `tc.lisp`.

```
(defun tc-n (n s) ...)
(zfn tc-fn (s) ...)
(defun tc (s) ...)
```

Outline

Outline

ZIFY INTRODUCTION: REVISITING `fib`

"Zify" rhymes with "reify" — it turns a unary ACL2 function into a ZF function (set of ordered pairs).

Look at the `fib` example in `zify.lisp`.

```
:trans1 (zify zfib fib :dom (omega) :ran (omega))
```

Below is a key part of the zify call above, informally:
$\texttt{(zfib)} = \{\langle p_1, p_2\rangle \in \omega \times \omega : p_2 = \texttt{fib}(p_1)\}$.

```
(zsub zfib ()
      p
      (prod2 (omega) (omega))
      (equal (cdr p) (fib (car p))))
```

ZIFY EXAMPLE: MIRROR (1)

For `zify`, the defaults for `:dom` and `:ran` — the domain and image (range) — are `(acl2)`, the set of good ACL2 objects.

```
(defun mirror (x)
  (cond ((atom x) x)
        (t (cons (mirror (cdr x))
                 (mirror (car x))))))
(prove-acl2p mirror) ; (ACL2P X) ⟹ (ACL2P (MIRROR X))
(zify zmirror mirror)
```

`Prove-acl2p` proves that a given function preserves `acl2p` ("good ACL2 object"). See file `prove-acl2p.lisp`.

- `:trans* t (prove-acl2p mirror)`
- **TIP**: Use `:trans*` instead of `:trans1` when `make-event` is involved in the expansion.

ZIFY EXAMPLE: MIRROR (2)

Now we can *attempt to* prove:

```
(thm (equal (apply (zmirror) '((a . b) . (c . d)))
            '((d . c) . (b . a))))
```

Fix it using `thmz` (and show `zify-prop`):

```
(thmz (equal (apply (zmirror) '((a . b) . (c . d)))
             '((d . c) . (b . a)))
      :props (zify-prop acl2$prop v$prop zmirror$prop))
```

`Zify*`

In ZF, every function is unary... (Why?)
because it is a set of ordered pairs $\langle x, y \rangle$.

`Zify*` is a variant of `zify` that can convert arbitrary-arity ACL2 functions to set-theoretic functions.

- The idea is to get a unary function that maps arglists to values.

You can see `zify.lisp` for a few examples, but time permitting we'll look at `foldr` in `foldr.lisp`.

Let's take a quick look at the book, `foldr.lisp`, up through:

```
(thmz
 (implies (acl2p lst)
          (equal (foldr lst (zbinary-*) 1)
                 (timeslist lst)))
 :props (foldr-prop zbinary-*$prop acl2$prop))

(thmz
 (implies (acl2p lst)
          (equal (foldr '(2 3 5) (zbinary-*) 1)
                 30))
 :props (foldr-prop zbinary-*$prop acl2$prop))
```

Outline

Outline

Cantor's Theorem

See `cantor.lisp` for a straightforward adaptation of the formalization and proof on Wikipedia.

Let's take a quick look — you can read the comments and events if interested in details.

- Note the natural use of `zsub` to follow the Wikipedia proof.
- **TIP**: Note the use of `minimal-theory` for control of the proof.
- **TIP**: It's OK to leave proof-builder `:instructions` when they're easily maintainable.

The Schröder-Bernstein Theorem

- If there is an injective function from A to B and also one from B to A, then there is a bijection from A to B.
- Based on Grant Jurgensen's ACL2 formalization
- We'll take a quick look at `schroeder-bernstein.lisp`.
  - Key idea: Zify the bijection provided by Grant's result.
  - Slight wart: Events need to support the `:prop`, `fun-bij`, introduced by that `zify` call.
- **TIP**: Locally included book `*-support.lisp` has ugly details (a technique used earlier in the rtl books and elsewhere).
- **TIP**: Hand proofs can be helpful; see `schroeder-bernstein-main-2-2` in `schroeder-bernstein-support.lisp`.

Outline

Outline

Future Work (Highly Incomplete List!)

- Transfinite recursion, e.g., V_α for all ordinals α
- Cardinals, cardinality (in progress)
- Higher-order applications (e.g., temporal logics)
- Tool improvements, e.g., let `zify` return `:REDUNDANT`
- More automation
  - ACL2 modification for parity-based rewriting (or maybe use existing clause-processor?)
  - Quantifier instantiation (maybe Dave Greve's stuff?)
  - Automated functional instantiation (maybe Joosten et al.'s 2013 workshop paper on `instance-of-defspec`)
- Prove correctness for the embedding of ACL2 into ZFG.
- More set theory
  - ω_1 (soon; should be easy using Cantor's theorem)
  - Cofinality, closed unbounded subsets, stationary sets
  - Mostowski collapse
  - Independence results
  - ...
- Other math, e.g., basic topology

WRAPPING UP

Possible PhD dissertation topic(s)?
Collaborators?

Thanks to Eric Smith and Kestrel Institute for recording and posting these talks.

Thank you for your attention!