ACL2 Best Practices

Recursion & Induction
Guest lecturer – Sol Swords (Arm Inc.)
March 13, 2025

Available at: http://bit.ly/3XSGPMj

Theory Management

- ACL2 quickly gets unmanageable when used with a disorganized set of rules (includes rewrites, function definitions, etc.)
- Especially important to disable function definitions when you're not expecting to reason directly about them
    - `defund`
    - `(in-theory (disable my-fndef))`
- When adding a theorem, think about how it will function as a rule—if not generally useful, disable it
    - `defthmd`
    - Or use `:rule-classes nil` to make it not a rule at all

Useful pattern for defining new functions

```
(encapsulate nil
  (defund my-function (x) ...)
  (local (in-theory (enable my-function)))
  (local (defthm local-lemma ...))
  (defthm exported-generally-useful-theorem ...)
  (defthmd exported-specifically-useful-theorem ...))
```

Abbreviation for the same pattern

```
(include-book “xdoc/top” :dir :system)

(defsection my-function      ;; nicer to look at
  (defund my-function (x) ...)
  (local (in-theory (enable my-function)))
  (local (defthm local-lemma ...))
  (defthm exported-generally-useful-theorem ...)
  (defthmd exported-specifically-useful-theorem ...))
```

Abbreviation for the same pattern

```
(include-book “std/util/define” :dir :system)

(define my-function (x)
   ;; automatically disabled non-locally and enabled locally
   ... ;; function body
  ///
  (local (defthm local-lemma ...))
  (defthm exported-generally-useful-theorem ...)
  (defthmd exported-specifically-useful-theorem ...))
```

Hints

- Changes to underlying definitions/rules cause hints to fail more often than a good rewriting theory
- Subgoal hints are particularly fragile
  - High probability that a tweak to a function definition will make it so something that happened on Subgoal *1/2.3.5’’’ now instead happens at Subgoal *1/3.3.4’’
  - Instead use “Goal” hints and `stable-under-simplificationp` hints – next slide
- A few generally useful classes of hints—others are for very specific, rare circumstances
  - :in-theory, :induct, :do-not-induct, :expand
  - :use (usually only at “Goal”)
- A good alternative to fragile hint structures in many cases: prove a special-purpose, local rule just for your particular situation

Reasonable use of hints

```
(encapsulate nil
  (local (defthm my-special-purpose-rule ...))
  (local (in-theory (e/d ((:i foo))
                         (a-conflicting-rule))))
  (defthm my-thm-about-foo
    ...
    :hints ((“goal” :induct (foo x y)
              :do-not-induct t        ;; no sub-inductions
              :expand ((foo x y)))
            (and stable-under-simplificationp
                 ‘(:in-theory (enable a-particular-rule))))))
```

Books

- Lots of libraries are available in the community books—look around
  - Github code search
- Use `include-book` with `:dir :system` or relative paths, no absolute paths
- Writing a certifiable book:
  - First form must be `(in-package ...)`
  - Allowed forms: `include-book, defun, defthm, defconst, defmacro, encapsulate, local, progn`
  - Macros / `make-events` that expand to the above
- Use of `local` in books is good but watch out for local incompatibility

Local incompatibility

```
(in-package “ACL2”)
(local (defun my-local-function (x) ...))
(defthm my-exported-theorem
   ;; local incompatibility - my-local-function not defined here
   (true-listp (my-local-function x)))
```

Another local incompatibility

```
(in-package “ACL2”)
(local (include-book “ihs/logops-lemmas” :dir :system))
(defthm a-theorem
   ;; local incompatibility - loghead not defined here
   (natp (loghead n x)))
```

Build system for books

- All books included must be certified before the book that includes them can be certified - can be a big dependency graph
- `acl2/books/build/cert.pl` automatically generates dependencies and certifies books in the right order using `make`
- Parallel builds with `-j 8` (whatever your machine can manage) recommended—lots of parallelism available
- Dependency analysis is via line-by-line scan so broken lines can fool it

```
(include-book  ;; don’t do this unless you want to fool the build system
  “my-book”)
```

Rule-classes

- `:rewrite` — general purpose workhorse
  - Best all round — try to make a good rewrite rule before considering another rule class for your theorem
- `:linear` — linear arithmetic, probably what you want if your conclusion is one of `<`, `>`, `<=`, `>=`
  - Triggers on some linear subterm of the inequality – e.g., in:

    ```
    (< (+ (* 2 (foo)) (- (* 3 (bar)))) (* 1/2 (baz)))
    ```

    Might choose to trigger on `(foo)`, `(bar)`, or `(baz)` — need to consider free variables
- `:type-prescription` — proves something is in some subset of the built-in types such as symbol, string, cons, positive integer, non-integer rational, etc. — see doc topic `type-set`
  - Best if no hypotheses — only relieved by type reasoning, not full simplification.
  - E.g. if a function always returns a natural number, good to have a type-prescription rule
  - If it returns a natural number when its input is greater than 5, a type-prescription rule will probably only be useful when you have input > 5 as an explicit assumption or forward-chaining result

More rule classes

- `:forward-chaining` — adds something to set of assumptions when some trigger term is assumed
    - Default trigger is first hypothesis
    - Hypotheses are only relieved by type reasoning/execution
    - Very special-purpose — when abused, can uselessly slow down the prover a lot
- `:compound-recognizer` – associates a user predicate with built-in types, see e.g. `natp-compound-recognizer`
- `:equivalence, :congruence, :refinement` — for user-defined equivalence classes
- `:definition` – alternative function definition, like rewrite but integrates with :expand hints and uses different heuristics for recursive expansion
- `:meta, :clause-processor` – install custom simplifiers/proof routines