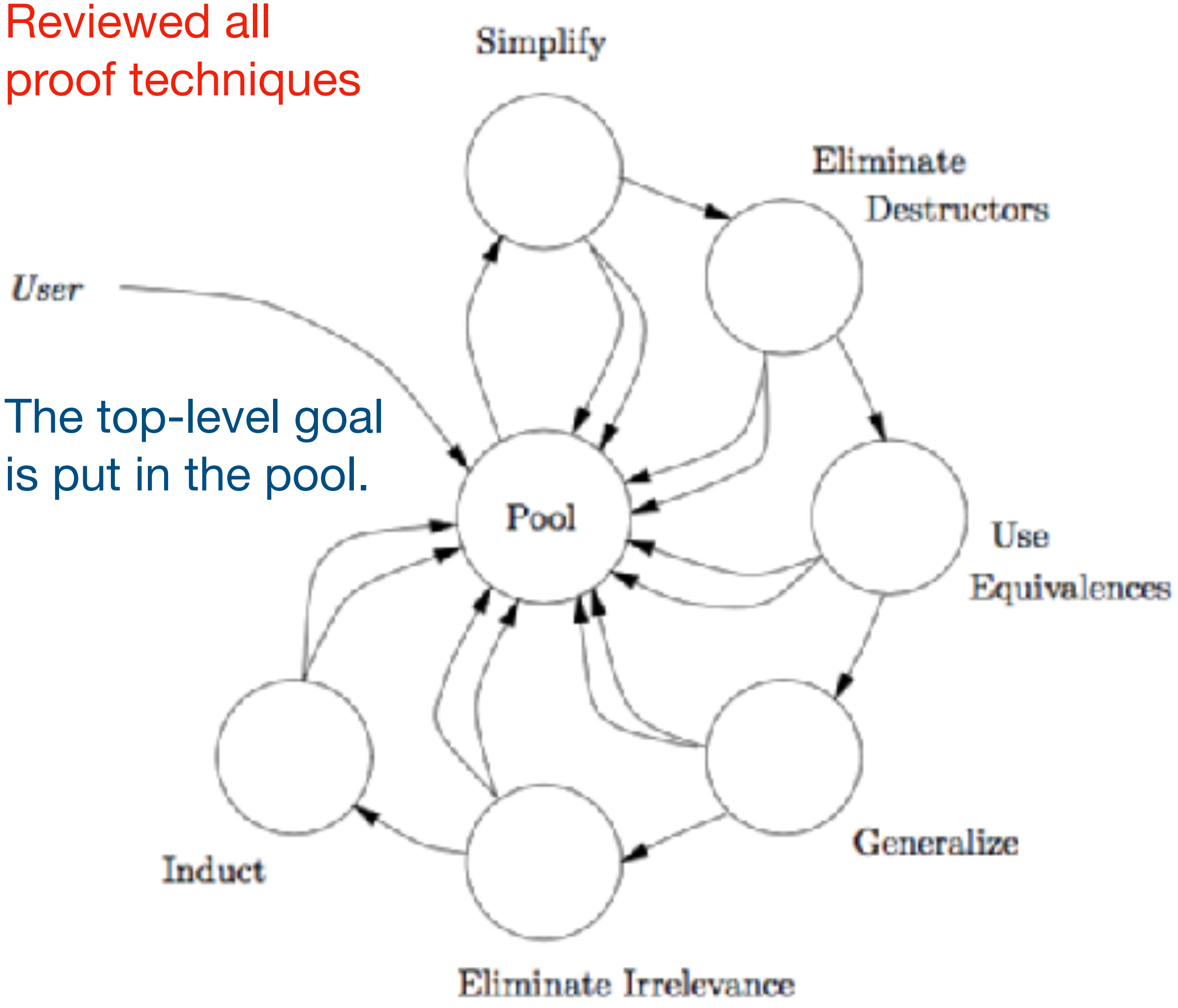
Lecture 8

Pete Manolios
Northeastern

Organization of ACL2

Reviewed all proof techniques



The top-level goal is put in the pool.

When a formula is drawn out, it is passed to proof techniques until one applies.

The draw is orchestrated that we do not try to prove a subgoal by induction until we have processed every subgoal produced by the last induction.

Slides by Pete Manolios for CS4820

Simplification in Detail

- Simplification is the heart of the theorem prover. It:
  - applies propositional calculus, equality, and linear arithmetic decision procedures,
  - uses type information and forward chaining rules to construct a "context" describing the assumptions of each subterm,
  - rewrites each subterm in the appropriate context, using definitions, conditional rewrite rules, and metafunctions,
  - use propositional calculus normalization to convert the resulting formula to an equivalent set of formulas, reduce the set under subsumption, and deposit the surviving formulas back in the pool.
- We'll discuss each of the four steps in the order in which they occur.
- First we discuss equivalence relations and congruence rules, which are fundamental to several aspects of the simplifier.

Congruence-Based Reasoning

- General form of substitution of equals for equals based on the ideas of user-defined equivalence relations and congruence rules.
- Consider: `(=> (set-equal x y)  (iff (member e x) (member e y)))`
- This congruence rule allows the substitution of `set-equals` for `set-equals`, in the second argument of member expressions, while preserving `iff`.
- Use `defequiv` to identify equivalence relations.
- Congruence rules: `(=> (equiv1 x y) (equiv2 (f...x...) (f...y...)))` where `equiv1` and `equiv2` are known equivalence relations.
- Use `defcong` to prove congruence rules.
- The rule allows `equiv1` substitution into f while preserving `equiv2`. ACL2 justifies deep substitutions by chaining together congruence rules, starting from a rule that preserves `iff` (propositional equivalence).

Decision Procedures

- When a formula is given to the simplifier three decision procedures are applied.
- Propositional Calculus based on the normalization of `if` expressions
    - Propositional connectives are expanded in terms of `if`
    - the `if` terms are distributed, so `(f (if a b c))` becomes `(if a (f b) (f c))` and `(if (if a b c) x y)` becomes `(if a (if b x y) (if c x y))`
    - the resulting tree is explored to determine whether every reachable tip is non-`nil`.
- Congruence Closure: use the context to compute equivalence classes, choose a representative per equivalence class, and substitute that representative for all members of the class. Repeat until fixpoint is reached.
- Rational linear arithmetic: linear data base contains all inequalities `(<,<=, >=,>,=)` relevant to conjecture, where function applications other than sums, differences, and products with constants are treated as variables.
    - *Linear rules* are theorems that concludes with an inequality. If an instance of one of the terms in the inequality arises in the linear data base, the rule is instantiated

Context

- Assume the formula to which the simplifier is applied is of the form `(=> (and p1 ... pn) q)`. The `pi` are the hypotheses and `q` is the conclusion.
- After decision procedures, the simplifier will rewrite each hypothesis and then the conclusion.
- Rewriting is done in a context that specifies what is assumed true.
  - For the conclusion, we assume all of the hypotheses.
  - For a hypothesis, we assume the other hypotheses and the negation of the conclusion.
- The context actually consists of two kinds of information: arithmetic and type theoretic.
  - Arithmetic inequalities from the assumptions and linear rules provide arithmetic information.
  - Type theoretic information: type algorithm, type-prescription & compound-recognizer rules.

Type-Theoretic Context

- *Type-prescription* rules allow you to inform the type algorithm of the type of the output produced by a function.
  - E.g., `(=> (^ (tlp a) (tlp b)) (tlp (app a b)))` allows the type algorithm to deduce the type of `(app a b)`.
- *Compound-recognizer rules* are applicable to Boolean-valued functions of one argument (recognizers).
  - E.g., `(=> (primep x) (posp x))` allows ACL2 to deduce type information about `x`.
- *Forward chaining rules*: any theorem
  - E.g., `(=> (and p1 . . . pn) q)`, where `p1` is the default trigger term (you can specify the trigger terms).
  - If an instance of the trigger occurs in the context and the `pi` are all true in the context, then `q` is added to the context.

Tau System

- Tau rules extend ACL2's type checker.
- The tau system is only tried when subgoals first enter the waterfall and when they are stable under simplification.
- Supports many kinds of rules, including
  - Simple: `(=> (p v) (q v))`
  - Conjunctive: `(=> (and (p1 v)...(pk v)) (q v))`
  - Signature: `(=> (and (p1 x1) (p2 x2)...)  (q (fn x1 x2...)))`
  - Eval, Signature Form 2, Bounder, Big Switch, MV-NTH Synonym, etc.
- p, q, p1, etc., denote monadic Boolean-valued function symbols, or equalities where one argument is constant, arithmetic comparisons in which one argument is a constant, or the negations of such terms.

Rewriter: High-Level Overview

- Variable & constants rewrite to themselves
- `(f a1 ... an)`: (*target*) In most cases, rewrite `ai`, to get `ai’` and rewrite `(f a1’ ... an’)` (inside-out)
- Special case(s): if `f` is `if`, rewrite the test, `a1`, to `a1’`; then rewrite `a2` and/or `a3` depending on whether we can establish if `a1’` is `nil`
- `(f a1’ ... an’)`: Consider all rules derived from axioms, definitions, theorems in reverse chronological order.
- Apply the first that fires & repeat
- All of this happens in simplification
- There is a rich underlying theory of term-rewriting

Rewrite Rules

- Rewrite rules are of the form:
  `(=> (and h1 ... hk) (equal (f b1 ... bn) rhs))`
- The definition of `f` is of this form (hyps are input contracts)
- A theorem concluding with `(not (p...))` is considered to conclude with `(iff (p...) nil)`
- A theorem concluding with `(p ...)`, where `p` is not a known equivalence relation and is not “`not`,” is considered to conclude with `(iff (p...) t)`
- Rule causes the rewriter to replace instances of *pattern* `(f b1 ... bn)` with the corresponding instance of `rhs` when they fire

Rewrite Rules

- Rewrite rule: `(=> (and h1 ... hk) (equal (f b1 ... bn) rhs))`
- Rule causes the rewriter to replace instances of *pattern* `(f b1 ... bn)` with the corresponding instance of `rhs` when they fire
- If we can instantiate variables in the pattern so that the pattern matches the target to get, say
  `(=> (and h1’ ... hk’) (equal (f a1’ ... an’) rhs’))`
- We try to apply the rule, by establishing its hypotheses
- Backchaining: Rewriting is used recursively to establish each hypothesis in the order in which they appear
- If successful, recursively rewrite `rhs’` to get `rhs’’`
- Certain heuristic checks are used to prevent some loops
- Finally, if certain heuristics approve of `rhs’’`, we say the rule fires and the result is `rhs’’`. This result replaces the target term.

Special Hypotheses

- `pi` is an arithmetic inequality, say `(< u v)`: the two arguments are rewritten, to `u′` and `v′`, and then the linear arithmetic decision procedure is applied to `(< u′ v′)`.
- An instantiated hypothesis contains free variables (e.g., transitivity). The rewriter looks for a binding of the free variables that make the hypothesis true. See set-match-free-default, which can be set to :once, :all, etc. Backtracking can occur.
- An instantiated hypothesis is of one of three forms:
  - `(syntaxp p)` always returns `t`. But when the rewriter encounters such a hypothesis it evaluates the form inside the `syntaxp` to decide whether the rule should fire.
  - `(force p)` is defined as the identity function. When the rewriter finds a hyp marked with force, it tries to establish it as above and if that fails it assumes hyp and goes on. These proofs are, by default, delayed until the successful completion of the main goal, using all the power of the theorem prover.
  - `(case-split p)` is a variant of force. When a hypothesis has the form `(case-split hyp)` it is logically equivalent to hyp. If ACL2 attempts to apply the rule but cannot establish the instance of hyp holds, it considers the hyp true anyhow, but creates a subgoal in which the instance of hyp is assumed false.

Heuristic Checks

- A rule for a function definition or *definition rule*, corresponds to expanding a call of the function. If the definition is recursive, we want to avoid looping: the rewriter will not fire the rule if the rewritten `rhs`, `rhs''`, fails certain tests.
  - One test permitting firing is that the arguments to the rewritten recursive call already appear in the formula being proved by the simplifier.
  - Another test permitting the firing is that the arguments be symbolically simpler.
- For rules like `(== (f x y) (f y x))` that permute arguments to a function, care is taken not to loop forever. Essentially, the system uses permutative rules only to swap arguments into "alphabetical" order.
- The rewriter just does what you tell it to do with your rewrite rules. If you tell it to loop forever, by rewriting `a` to `b`, `b` to `c`, and `c` to `a`, then it will loop forever, or as long as the resources of time and memory allow.

Normalization & Subsumption

- Assume the simplifier is working on `(=> (and p1 . . . pk) q)`, by rewriting the parts, and it has just rewritten `pk`. Suppose the result is a term that involves an if-expressions, say the result is `(p (if a b c))`. Then if normalization occurs.
- The simplifier tries to clean up the set of formulas.
  - For example, if one formula is `(=> p q)` and another is `(=> (and p r) q)`, then clearly we just prove the former.
  - If one formula is `(=> (and p r) q)` and another is `(=> (and p (not r)) q)`, then we just prove `(=> p q)`.
- If the result of subsumption/replacement is a set containing the input formula, then the simplifier passes the formula to dest elim.
- If the result is the empty set of formulas, then the simplifier proved the input formula.
- Otherwise, the simplifier deposits each of the formulas into the pool.

Driving ACL2

- You may begin to get the idea that the machine does everything for you. This is not true. A more accurate view is that the machine is a proof assistant that fills in the gaps in your "proofs." These gaps can be huge. When the system fails to follow your reasoning, you can use your knowledge of the mechanization to figure out what the system is missing.
- You are responsible for guiding ACL2 by proving the appropriate lemmas
- Rules generated by lemmas are rewrite rules
- You have to learn to program ACL2
- That involves building a mental model
- The ACL2 book advocates "the method"
- Once a proof attempt is started, one can interact with ACL2 only by interrupting the proof attempt

Proof Dags

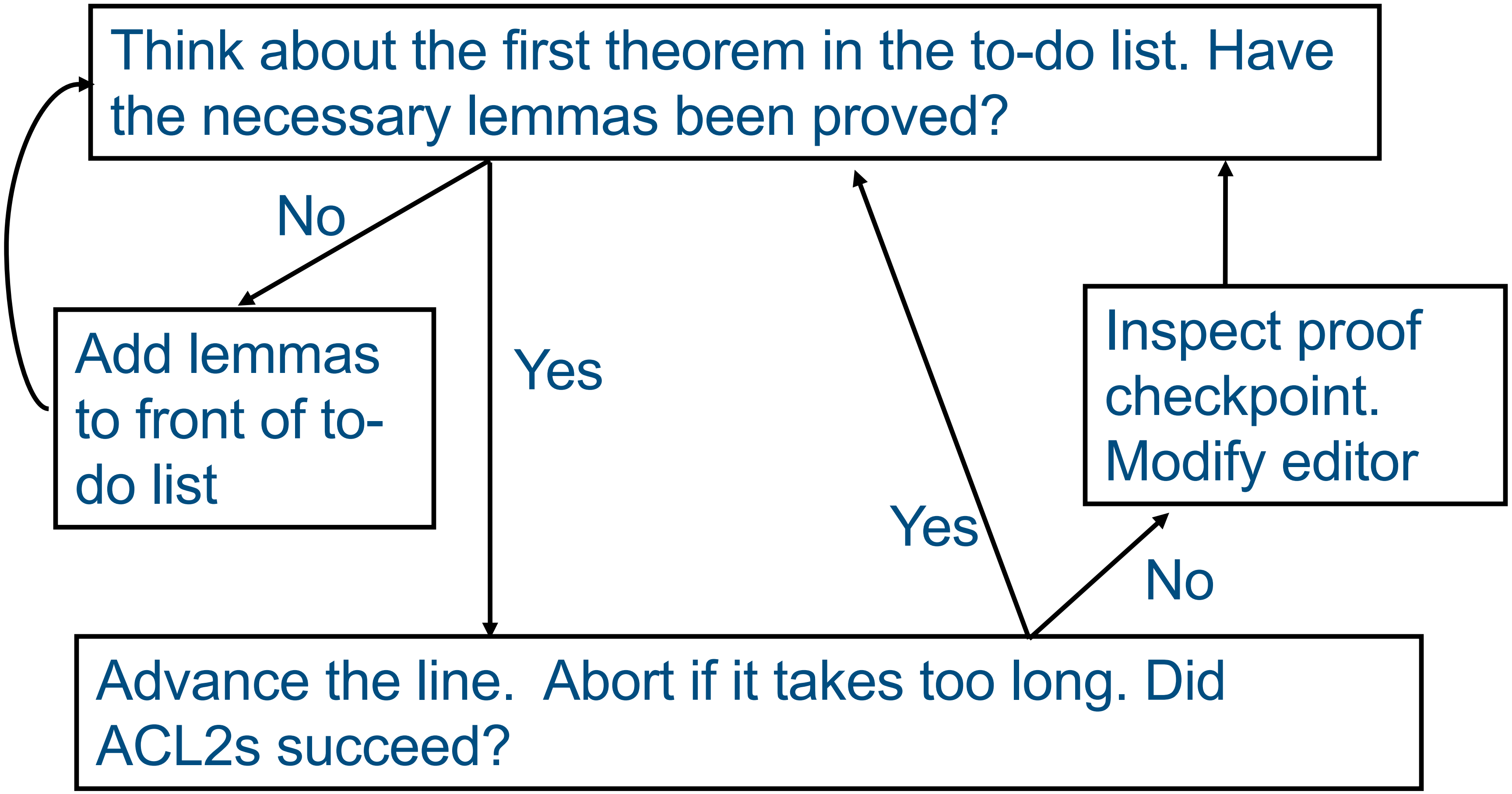
$$\begin{tikzcd}
 & & \text{Main} \arrow[dl] \arrow[d] \arrow[dr] & & \\
 & \text{A} \arrow[dl] \arrow[d] \arrow[dr] & \text{B} \arrow[d] & \text{C} \arrow[d] \arrow[dr] & \\
\text{A1} & \text{A2} \arrow[dl] \arrow[d] & \text{A3} \arrow[dl] & \text{C1} & \text{C2} \\
\text{A2a} & \text{A2b} & & &
\end{tikzcd}$$

- In the proof dag, every node corresponds to a lemma
- To lead ACL2 to a proof, you must prove every lemma (using a topological sort)
- As a practical matter, you may not have the dag
- The Method is a way of using ACL2 to discover it

The Method

- In ACL2s, we have an editor and a session
- Our code is in the editor, initially containing the main theorem
- When we are done, the editor will contain a topological sort of a proof dag
- During the project, the editor has a “line”
    - Above the line, is the done list: successful commands
    - Below line, to-do list: remaining commands

The Method


Think about the first theorem in the to-do list. Have the necessary lemmas been proved?
No
Add lemmas to front of to-do list
Yes
Yes
Inspect proof checkpoint. Modify editor
No
Advance the line. Abort if it takes too long. Did ACL2s succeed?

Theorem Proving Strategies

- ACL2 is really a programmable theorem prover
- Define a “normal” form & rules that assume/respect it
- Coming up with a rewrite strategy is key, e.g., if app associates to the left, then rules that associate it to the right are going to cause loops
- In addition to rewrite rules, there are *built-in-clause*, *clause-processor*, compound-recognizer, congruence, definition, elim, equivalence, forward-chaining, generalize, induction, linear, *meta*, *refinement*, tau-system, type-prescription, *type-set-inverter* and *well-founded-relation* rules and many options for controlling how they work
- You can also provide hints, including computed-hints, which allow you to write a program that computes hints based on the goal under consideration
- You can define your own theorem prover (meta rules), use external solvers (clause-processors), etc

Gaming

This part of the book is concerned with the mechanization of the logic. Our goal is to teach you how to use the theorem prover. We start, in this chapter, by sketching how the theorem prover works. Of course, knowing how something works—e.g., an automobile, a programming language, a violin—is quite different from knowing how to use it effectively.

As you read this chapter you may begin to get the idea that the machine does everything for you. This is not true. A more accurate view is that the machine is a proof assistant that fills in the gaps in your "proofs." These gaps can be huge. When the system fails to follow your reasoning, you can use your knowledge of the mechanization to figure out what the system is missing. But you may find that the machine's inability to fill in the gap is because your "proof" was simply wrong. Indeed, you may even find that the formula you "proved" is not even a theorem!

You may come to think of the proof process as a game. The theorem is the "opponent." It will use all legal means to dodge your weapons and squirm free of your traps and fences. It can hide amid innocuous detail, shatter into a swarm of subproblems, or stand crystalline still and shimmering in front of you, daring you to find a chink in its armor. In recognition of this view of theorem proving we have named this part of the book "Gaming." You will be hard pressed to find a more challenging game.

Kaufmann, Manolios, Moore in Computer-Aided Reasoning: An Approach

Rev-Rev
DEMO

Questions?