Lecture 6

Pete Manolios
Northeastern

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.
- Assume the formula to which the simplifier is applied is of the form (implies (and p_1 . . . p_n) q). The p_i are the hypotheses and q is the conclusion.
- First we discuss equivalence relations and congruence rules, which are fundamental to several aspects of the simplifier.
- Then we discuss each of the four steps in the order in which they occur.

Context

- 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., (true-listp (rev x)) allows the type algorithm to deduce that the type of (rev (app a b)) is either nil or a proper cons.
- *Compound-recognizer rules* are applicable to Boolean-valued functions of one argument (recognizers).
  - E.g., (implies (primep x) (posp x)) allows ACL2 to deduce type information about x.
- *Forward chaining rules*: a theorem of the form
  - E.g., (implies (and p_1 . . . p_n) q), where p_1 is the default trigger term (you can specify the trigger terms).
  - If an instance of the trigger occurs in the context and the p_i 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: (implies (p v) (q v))
  - Conjunctive: (implies (and (p_1 v) ... (p_k v)) (q v))
  - Signature: (implies (and (p_1 x_1) (p_2 x_2) ...) (q (fn x_1 x_2 ...)))
  - Eval, Signature Form 2, Bounder, Big Switch, MV-NTH Synonym, etc.
- p, q, p_1, 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 a_1 ... a_n): (*target*) In most cases, rewrite a_i, to get a'_i and rewrite (f a'_1 ... a'_n) (inside-out)
- Special case(s): if f is if, rewrite the test, a_1, to a'_1; then rewrite a_2 and/or a_3 depending on whether we can establish if a'_1 is nil
- (f a'_1 ... a'_n): 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:
  (implies (and h_1 ... h_k) (equal (f b_1 ... b_n) 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)
- Rules causes the rewriter to replace instances (f b_1 ... b_n) with the corresponding instance of rhs when they fire

Rewrite Rules

- Rewrite rule: (implies (and h_1 ... h_k) (equal (f b_1 ... b_n) rhs))
- Rule causes the rewriter to replace *pattern* (f b_1 ... b_n) 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
  (implies (and h'_1 ... h'_k) (equal (f a'_1 ... a'_n) 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

- p_i 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 (equal (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.

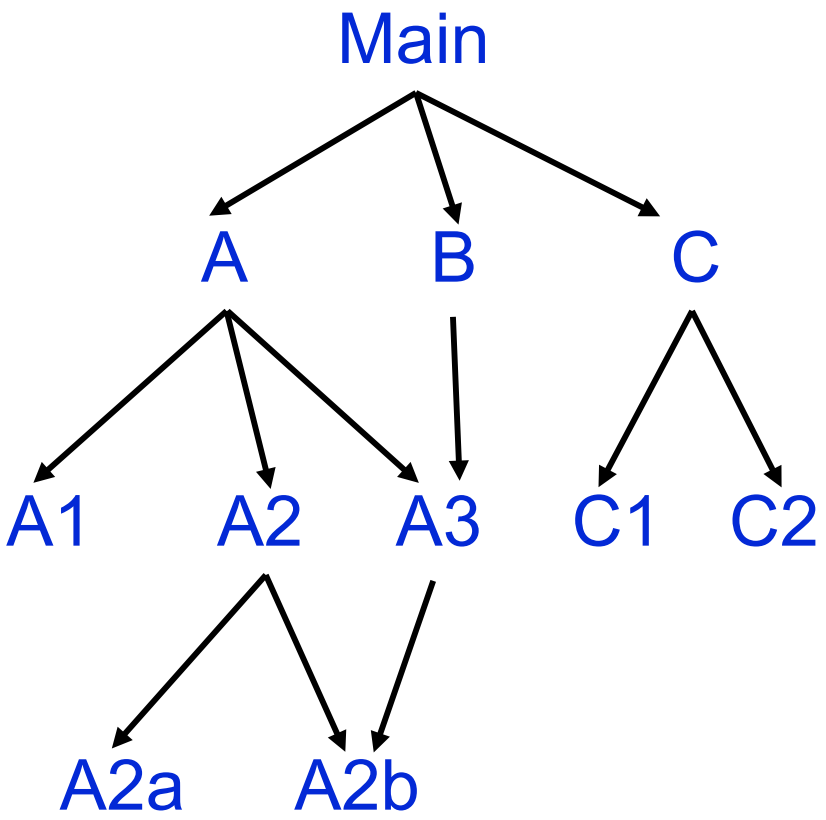
Normalization & Subsumption

- Assume the simplifier is working on (implies (and p_1 . . . p_k) q), by rewriting the parts, and it has just rewritten p_k. 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 (implies p q) and another is (implies (and p r) q), then clearly we just prove the former.
  - If one formula is (implies (and p r) q) and another is (implies (and p (not r)) q), then we just prove (implies 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 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 Tags

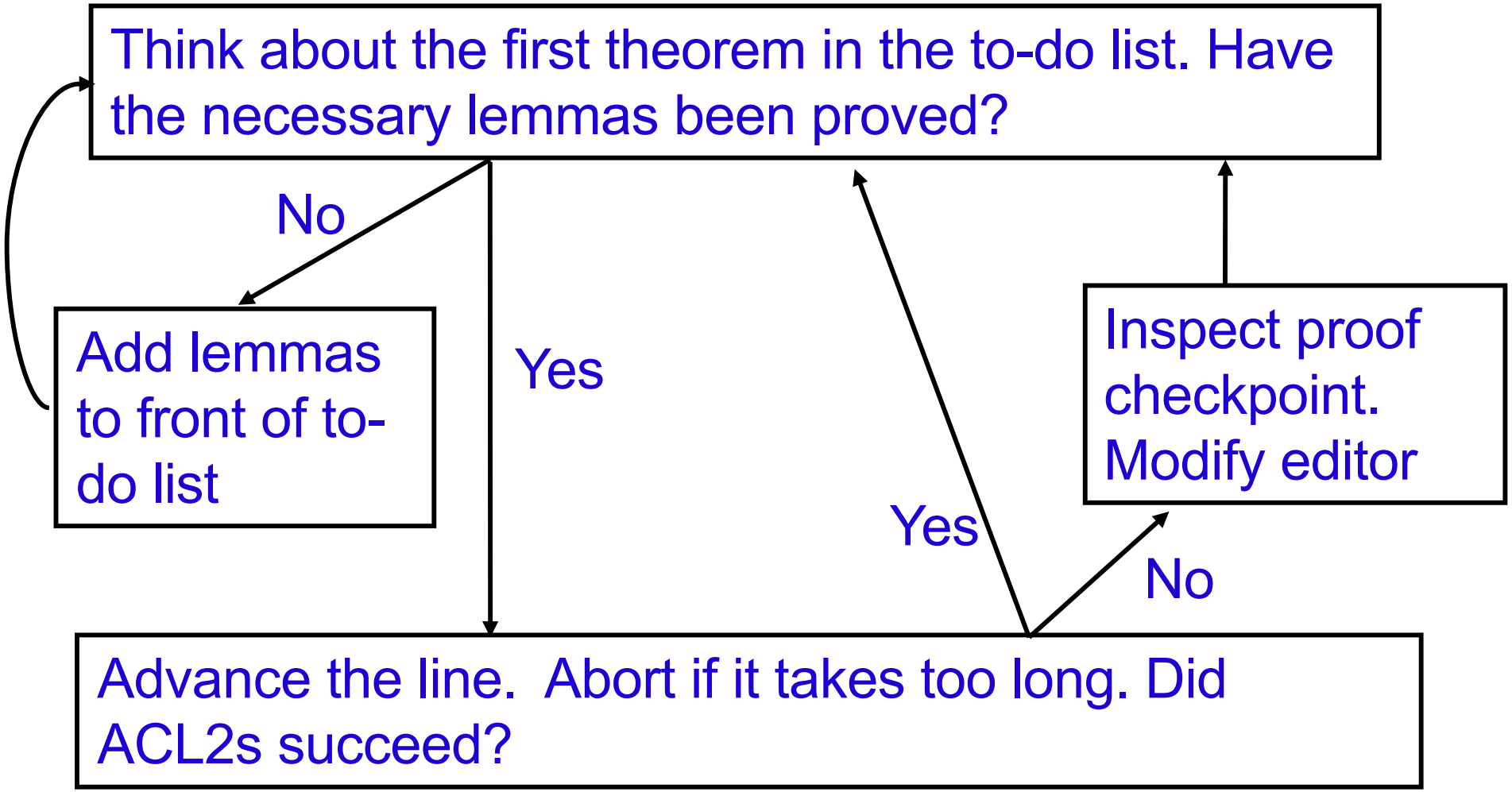


- In the proof dag, every node corresponds to a lemma
- To lead ACL2 to a proof, you must prove every lemma (using a topological sort)
- As a practical matter, you may not have the dag
- The Method is a way of using ACL2 to discover it

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
Inspect proof checkpoint. Modify editor
Yes
No
Advance the line. Abort if it takes too long. Did ACL2s succeed?

Failed Proofs and Generalization

- Focus on first subgoal that ACL2 cannot simplify which does not get proved
- Use the proof-tree “view” to select checkpoints
- Use the proof builder (use ACL2 as a proof checker)
- Consider defining and proving

```
(defun append (x y)
  (if (endp x)
      y
    (cons (car x) (append (cdr x) y))))


(defthm append-a
  (equal (append (append a a) a)
         (append a (append a a))))
```

- Generalization is key (as it is in all of math)

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

DEMO

Append/Reverse example

Defining rewrite rules

Rewrite strategies

Proof Builder (Proof Checker)

Examples