Tactics and Tracing
Background

• Goal
  – Proof checker for an ACL2-like logic
  – Small trusted core, self-verified extensions

• Current status
  – Proof checker and extensions defined in ACL2
  – Several extensions verified with ACL2
  – Trying to “port” these proofs to the core
Outline

- Tactics in an LCF-like prover
- Implementing a tactic system
- Demo: using tactics to prove theorems
- Rewriter tracing
LCF-like theorem provers

- Take a logic expressed with sequents

\[
\{ A_1, \ldots, A_n \} \vdash C
\]

- A \textbf{thm} represents a proof of a sequent

```cpp
class thm
{
    private formula_set assumptions;
    private formula conclusion;

    private thm() { }
    // ...
}
```
“Raw” proof construction

• Constructors correspond to inference rules

\[
\{ A_1, ..., A_n \} \vdash C \\
{ F, A_1, ..., A_n } \vdash C \quad \text{Weaken}
\]

class thm
{
    public static thm Weaken(thm orig, formula F) {
        ret = new thm();
        ret.assumptions = orig.assumptions.cons(F);
        ret.conclusion = orig.conclusion;
        return ret;
    }
    // ...
}
Goal-directed proof with tactics

- Goal-directed versus raw proof construction
- We represent goals as sequents

\[
\{ A_1, \ldots, A_n \} \vdash C
\]

```java
class goal {
    public formula_set assumptions;
    public formula conclusion;

    public goal() {}

    // ...
}
```
We can do backwards reasoning by inverting the inference rules

\[ \{ A_1, \ldots, A_n \} \vdash C \]

\[ \{ F, A_1, \ldots, A_n \} \vdash C \quad \text{Weaken} \]
We can do backwards reasoning by inverting the inference rules

\[
\{ A_1, \ldots, A_n \} \mid \vdash C
\]

\[
\{ F, A_1, \ldots, A_n \} \mid \vdash C
\]

Weaken

**goal** `strengthen_goal` (goal orig, formula `F`)

```cpp
{ goal ret = new goal();
  ret.assumptions = orig.assumptions.erase(F);
  ret.conclusion = orig.conclusion;
  return ret;
}
```
Recovering proofs

- Track the steps used to simplify a goal

Original goal: \( \{ A_1, A_2, A_3, C \} \vdash C \)

- Strengthen, \( A_1 \):
  \( \{ A_2, A_3, C \} \vdash C \)

- Strengthen, \( A_2 \):
  \( \{ A_3, C \} \vdash C \)

- Strengthen, \( A_3 \):
  \( \{ C \} \vdash C \)

Identity: true
Recovering proofs

- Then compile these steps back into a `thm`

Original goal \[ \{ A_1, A_2, A_3, C \} \vdash C \]

Strengthen, \( A_1 \) \[ \{ A_2, A_3, C \} \vdash C \] \( X_4 = \text{thm.Weaken}(X_3, A_1); \)

Strengthen, \( A_2 \) \[ \{ A_3, C \} \vdash C \] \( X_3 = \text{thm.Weaken}(X_2, A_2); \)

Strengthen, \( A_3 \) \[ \{ C \} \vdash C \] \( X_2 = \text{thm.Weaken}(X_1, A_3); \)

Identity \[ \text{true} \] \( X_1 = \text{thm.Identity}(C); \)
Simplifications may not be linear

And Introduction

\[ \{ \ldots \} \vdash A \]
\[ \{ \ldots \} \vdash B \]
\[ \{ \ldots \} \vdash A \wedge B \]
Goal lists consolidate the splits

Original Goal
{ ... } ‒ ...
Implementing a tactic system

- Representing goals
- Implementing reductions
- Tracking reductions as they are applied
- Reversing reductions to build the proof
- Interfacing with the user
Goal representation

- We use clauses instead of sequents
  - Sequent style \( \{ A_1, A_2, A_3 \} \vdash C \)
  - Clause style \( (\text{not } A_1) \lor (\text{not } A_2) \lor (\text{not } A_3) \lor C \)
Implementing reductions

• Reductions just simplify clause lists

(defund clause.remove-obvious-clauses (x)
  (declare (xargs :guard (logic.term-list-listp x)))
  (if (consp x)
      (if (clause.find-obvious-term (car x))
        (clause.remove-obvious-clauses (cdr x))
        (cons (car x) (clause.remove-obvious-clauses (cdr x))))
      nil))

• We have many reductions
  – Clause “cleaning”, splitting if expressions, generalizing, unconditional rewriting, ...
The initial skeleton just lists the original goals.

Other skeletons have the form

\[(\text{goals tacname extras history})\]

Where

- **Goals** are the clauses we still need to prove
- **Tacname** says which tactic we applied
- **History** is the skeleton we applied the tactic to
- ** Extras** store any additional information we'll want during proof construction
Tactic application = skeleton construction

Original Goals

\{ A \lor B \lor C \lor D \}  

Initial Skeleton

\{ A \lor B \lor C \lor D \}
Tactic application = skeleton construction

Original Goals
{ A \lor B \lor C \lor D }

Apply Cleanup
{ A \lor B \lor D }

Initial Skeleton
{ A \lor B \lor C \lor D } → { A \lor B \lor D }

“Cleanup”
No extras

History

Tactic application = skeleton construction

Original Goals

Apply Cleanup

Apply Split

Initial Skeleton

“Cleanup” No extras

“Split” No extras

History

History
Tactic application = skeleton construction

Original Goals \{ A \lor B \lor C \lor D \} → Initial Skeleton \{ A \lor B \lor C \lor D \} → "Cleanup" → "Split" → "Rewrite" → Rules → History

Apply Cleanup \{ A \lor B \lor D \} → "Cleanup" No extras

Apply Split \{ A \lor X \lor D, A \lor Y \lor D \} → "Split" No extras

Apply Rewrite \{ \} → "Rewrite"
An example of a tactic

- We write a function like this for each reduction

```lisp
(defun tactic.split-first-tac (x)  
  (declare (xargs :guard (tactic.skeletonp x)))  
  (let ((goals (tactic.skeleton->goals x)))  
    (if (not (consp goals))  
      (ACL2::cw "Split-first-tac failure: all clauses have already been proven."))  
    (let* ((clause1 (car goals))  
            (clause1-split (clause.split clause1))  
            (len (len clause1-split)))  
      (if (equal len 1)  
        (ACL2::cw "Split-first-tac failure: the clause cannot be split further."))  
        (tactic.extend-skeleton (app clause1-split (cdr goals)) 'split-first
                                len x))))))
```
Compiling skeletons into proofs

- Reductions must be reversible / justifiable

```
Subgoal 1
{ ... } ⊨ ...
```
```
Subgoal 2
{ ... } ⊨ ...
```
```
Subgoal 3
{ ... } ⊨ ...
```

```
Subgoal 1.1
{ ... } ⊨ ...
```
```
Subgoal 1.2
{ ... } ⊨ ...
```
```
Subgoal 2
{ ... } ⊨ ...
```
```
Subgoal 3
{ ... } ⊨ ...
```

“Split First”

```
Proof of Subgoal 1
```
```
Proof of Subgoal 2
```
```
Proof of Subgoal 3
```

“Split First Compiler”

```
Proof of Subgoal 1.1
```
```
Proof of Subgoal 1.2
```
```
Proof of Subgoal 2
```
```
Proof of Subgoal 3
```
Proving the original goals

- **Cleanup Compiler**
  - \( A \lor B \lor C \lor D \)

- **Split Compiler**
  - \( A \lor B \lor D \)

- **Rewrite Compiler**
  - \( A \lor X \lor D ; A \lor Y \lor D \)

- **Initial Skeleton**
  - \( A \lor B \lor C \lor D \)

- **“Cleanup”**
  - No extras
  - History

- **“Split”**
  - No extras
  - History

- **“Rewrite”**
  - Rules
  - History
An example of a tactic compiler

- We write a function like this for each reduction

```
defund tactic.split-first-compile (x proofs)
  (declare (xargs :guard (and (tactic.skeletonp x)
                               (tactic.split-first-okp x)
                               (logic.appeal-listp proofs)
                               (equal (clause.clause-list-formulas
                                       (tactic.skeleton->goals x))
                                      (logic.strip-conclusions proofs))))
  (let* ((history       (tactic.skeleton->history x))
          (old-goals     (tactic.skeleton->goals history))
          (clause1       (car old-goals))
          (len           (tactic.skeleton->extras x))
          (split-proofs  (firstn len proofs))
          (other-proofs  (restn len proofs))
          (clause1-proof (clause.split-bldr clause1 split-proofs)))
    (cons clause1-proof other-proofs)))
```
The tactic harness

Mush!
Bootstrapping with the tactic harness

- It manages an “implicit” proof skeleton
  - A new conjecture creates an initial skeleton
  - Applying tactics updates the skeleton
  - Completed skeletons can be compiled
  - The resulting proof is checked and saved to a file

- It also manages other state
  - Definitions, axioms, theorems, etc.
  - Flags and control variables for tactics
  - Rewrite rules and theories

- Demo
I originally tried the following system

Deficiencies:
- Rules have to be searched twice
- Builds proofs that are thrown away when hyps fail
- Terrible case-splitting in the builder's soundness proof
Trace-based approach

- The rewriter builds a record of what it did
- We can compile this record afterwards

- We can remember which rules we used
- We omit failed attempts to use rules
- Soundness proof is considerably easier
Trace representation

• Recursive maps (alists)
  – Method, the name for this type of step
  – Nhyps, the assumptions we used
  – Lhs, the term we rewrote
  – Rhs, the term we produced
  – Iffp, the context we rewrote under
  – Subtraces, a list of traces we built upon
  – Extras, anything extra we need for this kind of step
Recognizing valid trace steps

(defun rw.equiv-by-args-tracep (x)
  ;; [hyps ->] (equal a1 a1') = t
  ;; ...
  ;; [hyps ->] (equal an an') = t
  ;; -------------------------------------------------------
  ;; [hyps ->] (equiv (f a1 ... an) (f a1' ... an')) = t
  (declare (xargs :guard (rw.tracep x)))
  (let ((method   (rw.trace->method x))
        (nhyps    (rw.trace->nhyps x))
        (lhs      (rw.trace->lhs x))
        (rhs      (rw.trace->rhs x))
        (subtraces (rw.trace->subtraces x))
        (extras    (rw.trace->extras x)))
    (and (equal method 'equiv-by-args)
         (logic.functionp lhs)
         (logic.functionp rhs)
         (equal (logic.function-name lhs) (logic.function-name rhs))
         (equal (logic.function-args lhs) (rw.trace-list-lhses subtraces))
         (equal (logic.function-args rhs) (rw.trace-list-rhses subtraces))
         (all-equalp nil (rw.trace-list-iffps subtraces))
         (all-equalp nhyps (rw.trace-list-nhyps subtraces))
         (not extras)))

Recognizing full, valid traces

```lisp
(defun rw.trace-step-okp (x)
  (declare (xargs :guard (rw.tracep x)))
  (let ((method (rw.trace->method x)))
    (cond ((equal method 'fail) (rw.fail-tracep x))
          ((equal method 'transitivity) (rw.transitivity-tracep x))
          ((equal method 'equiv-by-args) (rw.equiv-by-args-tracep x))
          ((equal method 'lambda-equiv-by-args) (rw.lambda-equiv-by-args-tracep x))
          ((equal method 'beta-reduction) (rw.beta-reduction-tracep x))
          ((equal method 'ground) (rw.ground-tracep x))
          ((equal method 'urewrite-if-specialcase-nil) (rw.urewrite-if-specialcase-nil-tracep x))
          ((equal method 'urewrite-if-specialcase-t) (rw.urewrite-if-specialcase-t-tracep x))
          ((equal method 'urewrite-if-specialcase-same) (rw.urewrite-if-specialcase-same-tracep x))
          ((equal method 'urewrite-if-generalcase) (rw.urewrite-if-generalcase-tracep x))
          ((equal method 'urewrite-rule) (rw.urewrite-rule-tracep x))
          (t nil))))

(mutual-recursion
 (defund rw.trace-okp (x)
  (declare (xargs ...))
  (and (rw.trace-step-okp x)
       (rw.trace-list-okp (rw.trace->subtraces x))))

(defun rw.trace-list-okp (x)
  (declare (xargs ...))
  (if (consp x)
      (and (rw.trace-okp (car x))
           (rw.trace-list-okp (cdr x)))
      t)))
```
Compiling trace steps

(defun rw.compile-equiv-by-args-trace (x proofs)
  ;;  [hyps ->] (equal a1 a1') = t
  ;;  ...
  ;;  [hyps ->] (equal an an') = t
  ;;  -------------------------------------------------------
  ;;  [hyps ->] (equiv (f a1 ... an) (f a1' ... an')) = t
  (declare (xargs :guard (and (rw.tracep x)
                                (rw.equiv-by-args-tracep x)
                                (logic.appeal-listp proofs)
                                (equal (logic.strip-conclusions proofs)
                                       (rw.trace-list-formulas (rw.trace->subtraces x))))
                       :verify-guards nil))
  
  (let ((nhyps (rw.trace->nhyps x))
         (iffp  (rw.trace->iffp x))
         (name  (logic.function-name (rw.trace->lhs x))))
    (if (consp nhyps)
        (let ((hyps-formula (clause.clause-formula nhyps)))
          (if iffp
              (build.disjoined-iff-from-equal
               (build.disjoined-equal-by-args name hyps-formula proofs))
              (build.disjoined-equal-by-args name hyps-formula proofs)))
        (if iffp
            (build.iff-from-equal (build.equal-by-args name proofs))
            (build.equal-by-args name proofs))))
Compiling full traces

(defund rw.compile-trace-step (x proofs) 
 (declare (xargs :guard (and (rw.tracep x) 
 (rw.trace-step-okp x) 
 (logic.appeal-listp proofs) 
 (equal (logic.strip-conclusions proofs) 
 (rw.trace-list-formulas (rw.trace->subtraces x))))
 :verify-guards nil))
(let ((method (rw.trace->method x)))
 (cond ((equal method 'fail)                         (rw.compile-fail-trace x))
 ((equal method 'transitivity)                 (rw.compile-transitivity-trace x proofs))
 ((equal method 'equiv-by-args)                (rw.compile-equiv-by-args-trace x proofs))
 ((equal method 'lambda-equiv-by-args)         (rw.compile-lambda-equiv-by-args-trace x proofs))
 ((equal method 'beta-reduction)               (rw.compile-beta-reduction-trace x))
 ((equal method 'ground)                       (rw.compile-ground-trace x))
 ((equal method 'urewrite-if-specialcase-nil)  (rw.compile-urewrite-if-specialcase-nil-trace x proofs))
 ((equal method 'urewrite-if-specialcase-t)    (rw.compile-urewrite-if-specialcase-t-trace x proofs))
 ((equal method 'urewrite-if-specialcase-same) (rw.compile-urewrite-if-specialcase-same-trace x proofs))
 ((equal method 'urewrite-if-generalcase)      (rw.compile-urewrite-if-generalcase-trace x proofs))
 ((equal method 'urewrite-rule)                (rw.compile-urewrite-rule-trace x))
 ;; Sneaky twiddle for hypless iff theorem
  (t t)))))

(mutual-recursion
 (defund rw.compile-trace (x)
 (declare (xargs ...))
 (rw.compile-trace-step x (rw.compile-trace-list (rw.trace->subtraces x))))
(declare (xargs ...))
(if (consp x)
   (cons (rw.compile-trace (car x))
         (rw.compile-trace-list (cdr x)))
   nil)))
Conclusions: current status

- System can read its own definitions
- Several tactics implemented
  - Clause splitting, basic clause cleaning, “use hints”
  - Unconditional rewriting with evaluation
  - Induction and generalization
- Preliminary tactic harness implemented
  - Make conjectures, using tactics, saving proofs
  - Manage theories (enable, disable, restrict rules)
- Proved some simple theorems with tactics
Conclusions: the road ahead

- Implement and integrate a conditional rewriter
- Apply the rewriter to continue bootstrapping
- Prove an extension sound
- Propose
- ...
- Profit!
# Project Timeline

## Prior projects

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## 2004

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Confusion</th>
<th>Proof trees</th>
<th>Other projects, internship</th>
<th>Comp. reflection</th>
<th>Provisional checking</th>
<th>Core prover</th>
<th>Initial derivations</th>
<th>Demo extensions</th>
<th>Simple rewriter</th>
<th>ACL2 talk (core)</th>
<th>Terms</th>
<th>Deduction law</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>Feb</td>
<td>Mar</td>
<td>Apr</td>
<td>Jun</td>
<td>Jul</td>
<td>Aug</td>
<td>Sep</td>
<td>Oct</td>
<td>Nov</td>
<td>Dec</td>
<td></td>
</tr>
</tbody>
</table>

## 2005

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IV&amp;V class</th>
<th>Logic class</th>
<th>Comp. reflection</th>
<th>Provisional checking</th>
<th>Core prover</th>
<th>Initial derivations</th>
<th>Demo extensions</th>
<th>Simple rewriter</th>
<th>ACL2 talk (core)</th>
<th>Terms</th>
<th>Deduction law</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>Feb</td>
<td>Mar</td>
<td>Apr</td>
<td>Jun</td>
<td>Jul</td>
<td>Aug</td>
<td>Sep</td>
<td>Oct</td>
<td>Nov</td>
<td>Dec</td>
</tr>
</tbody>
</table>

## 2006

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HW verification class</th>
<th>Proposal, reading</th>
<th>Comp. reflection</th>
<th>Provisional checking</th>
<th>Core prover</th>
<th>Initial derivations</th>
<th>Demo extensions</th>
<th>Simple rewriter</th>
<th>ACL2 talk (eval)</th>
<th>LCAR</th>
<th>Isabelle (oops)</th>
<th>ACL2 talk (Isabelle)</th>
<th>Early tactics</th>
<th>Claustify, irref.</th>
<th>Uncond rw. #2 (start)</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>Feb</td>
<td>Mar</td>
<td>Apr</td>
<td>Jun</td>
<td>Jul</td>
<td>Aug</td>
<td>Sep</td>
<td>Oct</td>
<td>Nov</td>
<td>Dec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## 2007

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Future</th>
<th>2007</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Evaluation**
- **Uncond. rw. #1**
- **Cond. rw. (oops)**
- **Gradest poster**
- **Cybertrust talk**
- **Simplifier (oops)**
- **Hons**
- **Fj4181, talk**
- **Martin Marek talk**
- **Propositional checking**
- **ACL2 talk (eval)**
- **LCAR**
- **Isabelle (oops)**
- **ACL2 talk (Isabelle)**
- **Early tactics**
- **Claustify, Irref.**
- **Uncond. rw. #2 (start)**
- **Matching**
- **Deferiv**
- **Uncond. rw. #2 (end)**
- **Puredoc**
- **Tracing for rewrite**
- **Clause overhauling**
- **Tactic harness**
- **Initial bootstrapping**
- **Nap-off-ten**
- **Proposal, reading**
- **Honors**
- **AC2 talk (Isabelle)**