Milawa

Adding a Computation Rule

Jared Davis

Recorded June, 2006

Outline

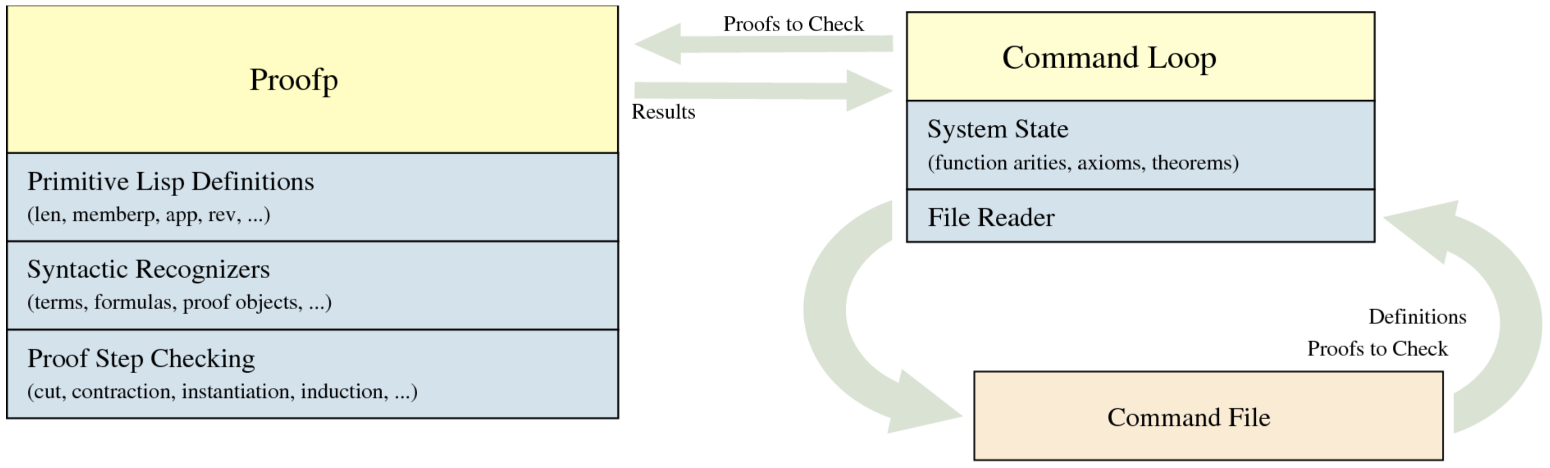
- Project overview
- The core proof checker
- Constructing proofs
- Extending the prover
- A computation extension
- ACL2 proof of soundness
- Bootstrapping

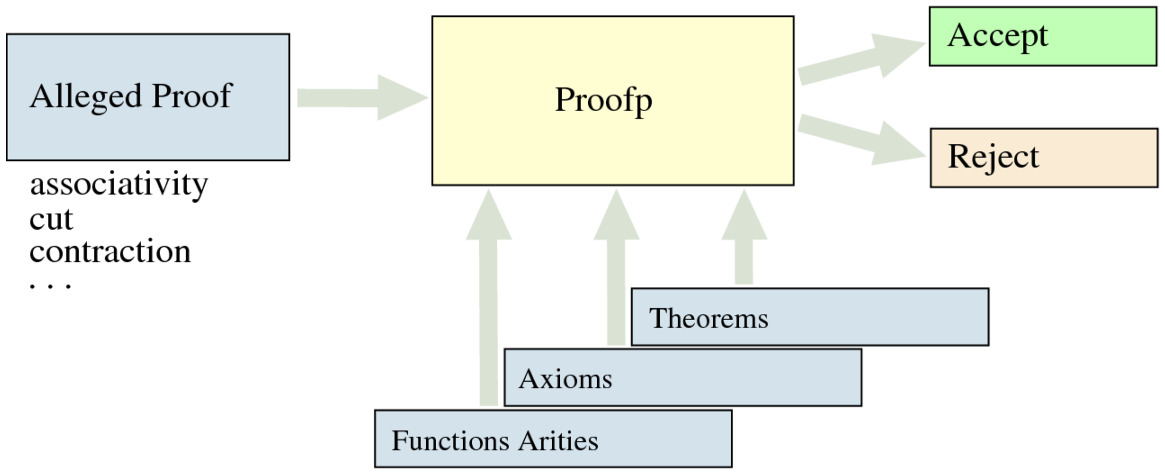
1. Project overview

- Build a capable prover we can trust
  - Start with a simple proof checker
  - Extend it with new capabilities
  - Prove the extensions are “sound”

- Proof checker is written in its own logic
  - It can reason about what it can prove
  - It can prove its own extensions are sound

The core program


Proofp
Primitive Lisp Definitions
(len, memberp, app, rev, ...)
Syntactic Recognizers
(terms, formulas, proof objects, ...)
Proof Step Checking
(cut, contraction, instantiation, induction, ...)
Proofs to Check
Results
Command Loop
System State
(function arities, axioms, theorems)
File Reader
Definitions
Proofs to Check
Command File



Extensions

Derived Rules of Inference
Propositional Rules
Equality Rules
Lambda Reduction Rules
The Deduction Law
The Tautology Theorem
Equivalence Substitution
Substitution of Equals for Equals

Heuristic Theorem Proving
Formula Compilation, Clausification
Calculation of Ground Terms
Primitive Type Reasoning
Conditional Term Rewriting
Linear Arithmetic Reasoning
Clause Simplifier

☆ this talk

Using ACL2 for sketches

- We can define proofp in ACL2
  - And use ACL2 to prove extensions sound

- Eventual goal: use the core proof checker
  - ACL2 proofs are sketches we can try to follow
  - We try to make them simpler to follow
    - No linear, type-prescription, forward-chaining, equivalence, congruence rules are used

2. The core proof checker

- Universe
  - Rationals, symbols, and (recursively) pairs

- Terms
  - Variables
  - Quoted constants
  - Function applications: $(f\ t_1 \ldots t_n)$
  - Lambdas: $((\lambda\ (x_1 \ldots x_n)\ B)\ t_1 \ldots t_n)$

Terms

```
(defun milawa-objectp (x) ...)
(defun variablep (x)      ...)
(defun constantp (x)      ...)


(mutual-recursion
  (defun term-structurep (x)      ...)
  (defun term-structure-listp (x) ...))

(mutual-recursion
  (defun termp (x atbl)      ...)
  (defun term-listp (x atbl) ...))


(defun fterm (fn args) ...)
(defun ftermp (x)      ...)
(defun fast-ftermp (x) ...)
(defun fname (x)       ...)
(defun fargs (x)       ...)


(defun lambda-term (formals body actuals) ...)
(defun lambda-termp (x)                   ...)
(defun fast-lambda-termp (x)              ...)
(defun lambda-formals (x)                 ...)
(defun lambda-body (x)                    ...)
(defun lambda-actuals (x)                 ...)
```

Formulas

- Equalities, negations, and disjunctions

  (:pequal t_1 t_2), (:pnot F), (:por F G)

```
(defun formula-structurep (x) ...)
(defun formulap (x atbl)      ...)

(defun fmtype (x)   (first x))

(defun pequal (a b) (list :pequal a b))
(defun =lhs   (x)   (second x))
(defun =rhs   (x)   (third x))

(defun pnot   (f)   (list :pnot f))
(defun ~arg   (x)   (second x))

(defun por    (f g) (list :por f g))
(defun vlhs   (x)   (second x))
(defun vrhs   (x)   (third x))
```

Proofs

- Trees of *appeals*
  - (method conclusion [subgoals] [extras])
  - “Applying method to subgoals yields conclusion”

```
(mutual-recursion
 (defun appeal-structurep (x)
   (and (true-listp x)
        (<= 2 (len x))
        (<= (len x) 4)
        (symbolp (first x))
        (formula-structurep (second x))
        (appeal-structure-listp (third x))))
 (defun appeal-structure-listp (x)
   (if (consp x)
       (and (appeal-structurep (car x))
            (appeal-structure-listp (cdr x)))
     (equal x nil))))

(defun get-method     (x) (first x))
(defun get-conclusion (x) (second x))
(defun get-subgoals   (x) (third x))
(defun get-extras     (x) (fourth x))
```

Checking proof steps

- Example: the contraction rule
  - Derive A from A v A

```
(defun contraction-okp (x)
  (let ((method     (get-method x))
        (conclusion (get-conclusion x))
        (subgoals   (get-subgoals x))
        (extras     (get-extras x)))
    (and (equal method :contraction)
         (equal extras nil)
         (equal (len subgoals) 1)
         (let* ((subgoal (get-conclusion (first subgoals))))
           (and (equal (fmtype subgoal) :por)
                (equal (vlhs subgoal) conclusion)
                (equal (vrhs subgoal) conclusion))))))
```

Checking proof steps (2)

- Example: the expansion rule
  - Derive B v A from A

```
(defun expansion-okp (x)
  (let ((method     (get-method x))
        (conclusion (get-conclusion x))
        (subgoals   (get-subgoals x))
        (extras     (get-extras x)))
    (and (equal method :expansion)
         (equal extras nil)
         (equal (len subgoals) 1)
         (let* ((subgoal (get-conclusion (first subgoals))))
           (and (equal (fmtype conclusion) :por)
                (equal (vrhs conclusion) subgoal))))))
```

Checking arbitrary proof steps

```
(defun appeal-provisionally-okp (x axioms thms atbl)
  (declare (xargs :guard (and (arity-tablep atbl)
                              (formula-listp axioms atbl)
                              (formula-listp thms atbl)
                              (appealp x atbl))))
  (case (get-method x)
    (:axiom                 (axiom-okp                x axioms))
    (:theorem               (theorem-okp              x thms))
    (:propositional-schema (propositional-schema-okp x))
    (:functional-equality  (functional-equality-okp  x atbl))
    (:lambda-equality      (lambda-equality-okp      x))
    (:expansion            (expansion-okp            x))
    (:contraction          (contraction-okp          x))
    (:associativity        (associativity-okp        x))
    (:cut                  (cut-okp                  x))
    (:instantiation        (instantiation-okp        x atbl))
    (:induction            (induction-okp            x atbl))
    (:base-eval            (base-eval-okp            x))
    ;; BOZO eventually add :reflection
    (otherwise             nil)))
```

Checking whole proofs

```
(mutual-recursion
 (defun **aux-proofp** (x axioms thms atbl)
   (declare (xargs :guard (and (arity-tablep atbl)
                               (formula-listp axioms atbl)
                               (formula-listp thms atbl)
                               (appealp x atbl))))
   (and (appeal-provisionally-okp x axioms thms atbl)
        (aux-proof-listp (get-subgoals x) axioms thms atbl)))
 (defun **aux-proof-listp** (xs axioms thms atbl)
   (declare (xargs :guard (and (arity-tablep atbl)
                               (formula-listp axioms atbl)
                               (formula-listp thms atbl)
                               (appeal-listp xs atbl))))
   (if (consp xs)
       (and (aux-proofp (car xs) axioms thms atbl)
            (aux-proof-listp (cdr xs) axioms thms atbl))
     (equal xs nil))))

(defun **proofp** (x axioms thms atbl)
  (declare (xargs :guard (and (arity-tablep atbl)
                              (formula-listp axioms atbl)
                              (formula-listp thms atbl))))
  (and (appealp x atbl)
       (aux-proofp x axioms thms atbl)))
```

3. Building proofs

- Primitive rules are too limited
- We write new functions to ease proof building
  - Note: proof sizes matter for bootstrapping
- Builders build new proofs from some inputs
  - Usually other proofs, formulas, or terms
  - These inputs need to be well formed

Primitive builders

```
(defun axiom (formula)
  (list :axiom formula))

(defun theorem (formula)
  (list :theorem formula))

(defun propositional-schema (a)
  (list :propositional-schema (por (pnot a) a)))

(defun contraction (a)
  (list :contraction (vrhs (get-conclusion a)) (list a)))

(defun cut (a b)
  (list :cut
        (por (vrhs (get-conclusion a))
             (vrhs (get-conclusion b)))
        (list a b)))

(defun instantiation (x sigma)
  (let* ((conclusion (get-conclusion x))
         (instance   (substitute-formula conclusion sigma)))
    (if (equal conclusion instance)
        x
      (list :instantiation instance (list x) sigma))))
```

Derived builders

- Example: commutativity of or
  - Derive B v A from A v B
  - “Cost” is 2 + x

1. A v B	Given
2. ~A v A	Propositional Axiom
3. B v A	Cut; 1,2

```
(defun commute-or-bldr (x)
  (let* ((or-a-b (get-conclusion x))
         (a      (vlhs or-a-b)))
    (cut x (propositional-schema a))))
```

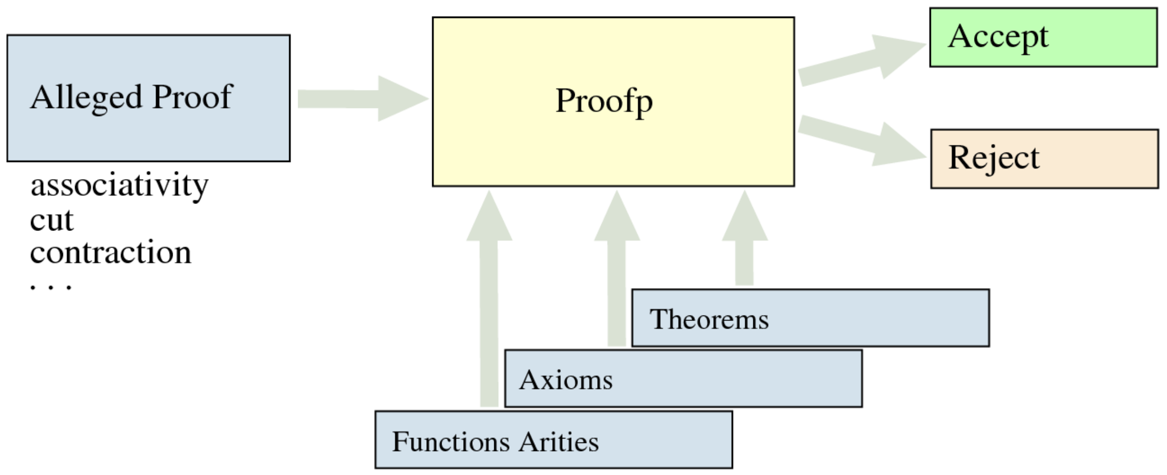
Nesting builders

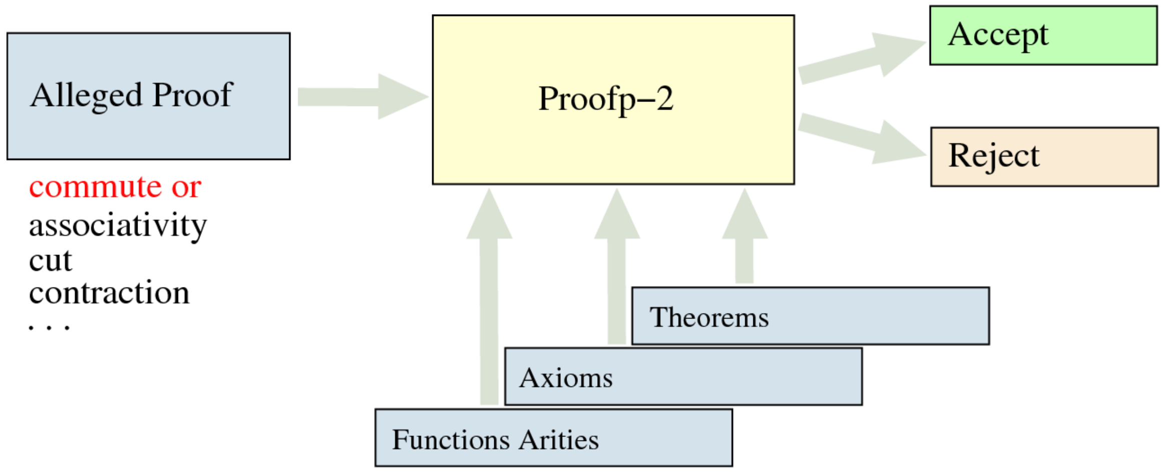
- We can use these builders freely
  - Note: costs are inherited!

```
(defun right-expansion-bldr (x b)
  (commute-or-bldr (expansion b x)))

(defun modus-ponens-bldr (x y)
  (let* ((or-not-a-b (get-conclusion y))
         (b          (vrhs or-not-a-b)))
    (contraction
     (cut (right-expansion-bldr x b)
          y))))

(defun right-associativity-bldr (x)
  (commute-or-bldr
   (associativity
    (commute-or-bldr
     (associativity
      (commute-or-bldr x))))))
```

- Tour of our constructors

4. Extending the prover





Writing an extended prover

```
(defun commute-or-okp (x)
  (let ((method     (get-method x))
        (conclusion (get-conclusion x))
        (subgoals   (get-subgoals x))
        (extras     (get-extras x)))
    (and (equal method :commute-or)
         (equal extras nil)
         (equal (len subgoals) 1)
         (let* ((subgoal (get-conclusion (first subgoals))))
           (and (equal (fmtype subgoal) :por)
                (equal (fmtype conclusion) :por)
                (equal (vlhs conclusion) (vrhs subgoal))
                (equal (vrhs conclusion) (vlhs subgoal)))))))

(defun appeal-provisionally-okp-2 (x axioms thms atbl)
  (case (get-method x)
    (:commute-or (commute-or-okp x))
    (otherwise   (appeal-provisionally-okp x axioms thms atbl))))

(mutual-recursion
 (defun aux-proofp-2 (x axioms thms atbl)
   (and (appeal-provisionally-okp-2 x axioms thms atbl)
        (aux-proof-listp-2 (get-subgoals x) axioms thms atbl)))
 (defun aux-proof-listp-2 (xs axioms thms atbl)
   (if (consp xs)
       (and (aux-proofp-2 (car xs) axioms thms atbl)
            (aux-proof-listp-2 (cdr xs) axioms thms atbl))
     (equal xs nil))))
```

Soundness of proofp-2

- Goal: show that proofp-2 is sound w.r.t. proofp

```
(defun-sk provablep (formula axioms thms atbl)
  (exists proof
          (and (proofp proof axioms thms atbl)
               (equal (get-conclusion proof) formula))))
```

- Statement of soundness (in ACL2)

```
(defthm provablep-when-proofp-2
  (implies (proofp-2 x axioms thms atbl)
           (provablep (get-conclusion x) axioms thms atbl)))
```

The proof in ACL2

- Boils down to the following lemma

```
(defthm **soundness-of-commute-or-okp**
  (implies (and (appealp x atbl)
                (commute-or-okp x)
                (provable-listp (strip-conclusions (get-subgoals x))
                                axioms thms atbl))
           (provablep (get-conclusion x) axioms thms atbl)))
```

- Proof sketch
  - Assume (commute-or-okp x). Then x concludes B v A from a subgoal which concludes A v B.
  - Assume (provable-listp ...). This means A v B is provable, so let p be a proof of A v B.
  - Now (commute-or-bldr p) is a proof which concludes B v A, so B v A is provable. Q.E.D.

5. A computation extension

Derived Rules of Inference
Propositional Rules
Equality Rules
Lambda Reduction Rules
The Deduction Law
The Tautology Theorem
Equivalence Substitution
Substitution of Equals for Equals

Heuristic Theorem Proving
Formula Compilation, Clausification
Calculation of Ground Terms ☆
Primitive Type Reasoning
Conditional Term Rewriting
Linear Arithmetic Reasoning
Clause Simplifier

- Want to be able to “evaluate away” ground terms that occur in proof attempts
  - Example, replace (fact 10) with 3,628,800

Base evaluation

- Infinitely many constants
- Base functions
  - if, equal, cons, consp, car, cdr, integerp, +, *, ...
- Base evaluation axiom schema
  - Derive (fn c_1 ... c_n) = result
    - Where fn is any base function
    - Where c_1 ... c_n are constants

Checking base evaluation

```
(defun initial-arity-table ()
  '((if . 3)
    (equal . 2)
    (consp . 1)
    (cons . 2)
    (car . 1)
    (cdr . 1)
    (symbolp . 1)
    (rationalp . 1)
    (integerp . 1)
    (binary-< . 2)
    (binary-+ . 2)
    (unary-- . 1)
    (binary-* . 2)
    (unary-/ . 1)))

(defun base-evaluablep (x)
  (and (ftermp x)
       (let ((fn (fname x))
             (args (fargs x)))
         (let ((entry (lookup fn (initial-arity-table))))
           (and entry
                (constant-listp args)
                (equal (len args) (cdr entry)))))))
```

Checking base evaluation (2)

```
(defun base-evaluator (x)
  (let ((fn   (fname x))
        (vals (strip-quotes (fargs x))))
    (list 'quote
          (cond ((equal fn 'if)
                 (if (first vals)
                     (second vals)
                   (third vals)))

                ((equal fn 'equal)
                 (equal (first vals)
                        (second vals)))

                ((equal fn 'consp)
                 (consp (first vals)))

                ((equal fn 'cons)
                 (cons (first vals)
                       (second vals)))

                ((equal fn 'car)
                 ;; this is the same as (car a), but guard-verifies.
                 (let ((a (first vals)))
                      (if (consp a)
                          (car a)
                   nil)))

                ...))
```

Checking base evaluation (3)

```
(defun base-eval-okp (x)
  (let ((method     (get-method x))
        (conclusion (get-conclusion x))
        (subgoals   (get-subgoals x))
        (extras     (get-extras x)))
    (and (equal method :base-eval)
         (equal subgoals nil)
         (equal extras nil)
         (equal (fmtype conclusion) :pequal)
         (let ((lhs (=lhs conclusion))
               (rhs (=rhs conclusion)))
           (and (base-evaluablep lhs)
                (equal (base-evaluator lhs) rhs))))))
```

- We also have a simple builder

```
(defun base-eval (term)
  (list :base-eval
        (pequal term (base-evaluator term))))
```

Generic evaluation

- We write (generic-evaluator term defs n)

– *term*	a ground term to evaluate
– *defs*	a list of function definitions
– *n*	a “stack depth” to enforce termination

- Which returns

– *nil*	if we can't evaluate the term
– *result*	a quoted constant, otherwise

Generic-evaluator

```
(defun generic-evaluator (term defs n)

  (cond

    ((zp n)
     ;; Failure: ran out of stack depth
     (ACL2::cw "Warning: insufficient n given to generic-evaluator~%"))

    ((constantp term)
     ;; SUCCESS: Constants evaluate to themselves.
     term)

    ((variablep term)
     ;; FAILURE: Not a ground term.
     nil)
```

Generic-evaluator (lambdas)

```
((fast-lambda-termp term)
 (let ((formals (lambda-formals term))
       (body    (lambda-body term))
       (actuals (lambda-actuals term)))

   ;; We eagerly evaluate the arguments to a lambda.
   (let ((results (generic-evaluator-list actuals defs n)))
     (and results
          ;; The arguments successfully evaluated.
          ;; Now substitute the results into the body.
          (generic-evaluator
           (substitute-term body (pair-lists formals (cdr results)))
           defs
           (+ -1 n))))))
```

Generic-evaluator (ifs)

```
((fast-ftermp term)

     (if (equal (fname term) 'if)
         ;; "if" must be handled specially with lazy evaluation.
         (let ((args (fargs term)))
           (and (equal (len args) 3)
                (let ((result (generic-evaluator (first args) defs n)))
                  (and result
                       ;; Test successfully evaluated, result is a quoted
                       ;; constant; follow the appropriate branch.
                       (if (second result)
                           (generic-evaluator (second args) defs n)
                         (generic-evaluator (third args) defs n))))))
```

Generic-evaluator (non-ifs)

```
;; Eagerly evaluate arguments to other functions
(let ((fn      (fname term))
      (results (generic-evaluator-list (fargs term) defs n)))
  (and results
       ;; The arguments successfully evaluated, results is (t (quote
       ;; val1) ... (quote valn)).  Let values be ('val1 ... 'valn).
       (let ((values (cdr results)))
         (if (memberp fn (domain (initial-arity-table)))
             ;; Preliminary function found.
             (and (equal (cdr (lookup fn (initial-arity-table)))
                         (len values))
                  (base-evaluator (fterm fn values)))

           ;; Non-preliminary function.  Attempt to look up its definition.
           (let* ((def (definition-list-lookup fn defs)))
             (and def
                  ;; Found the definition, extract the formals and body.
                  (let ((formals (fargs (=lhs def)))
                        (body    (=rhs def)))
                    (and (equal (len formals) (len values))
                         (generic-evaluator
                          (substitute-term body (pair-lists formals values))
                          defs
                          (+ -1 n))))))))))
```

6. ACL2 proof of soundness

- The main result
  - If (generic-evaluator term defs n) succeeds, then *term* = *result* is provable.

```
(defthm generic-evaluator-is-sound
  (implies (and [everything is well formed according to atbl]
                [all definitions are axioms]
                [certain primitive axioms are present]
                (generic-evaluator term defs n))
           (provablep (pequal term (generic-evaluator term defs n))
                      axioms thms atbl)))
```

- Proof strategy same as before
  - Define a generic-evaluator-bldr function to construct the proof
  - Observe that the function behaves correctly

Generic-evaluator-bldr

- Actually not hard to define
  - Follow the structure of generic-evaluator
  - There are five things to consider
    - Constants (trivial by reflexivity)
    - Lambda expressions
    - If expressions (i'll skip these)
    - Base functions
    - Defined functions

Lambda expressions

Goal: ((λ formals body) actuals) = result

Proof sketch.

1. Recursively prove actuals[i] = results[i] for all i.
2. ((λ formals body) actuals) = ((λ formals body) results) by the derived rule “λ pequal by args”
3. ((λ formals body) results) = body/[formals<-results] by beta reduction
4. Recursively prove body/[formals<-results] = result
5. Transitivity of = finishes the proof.

Base functions

Goal: (fn arg_1 ... arg_n) = result

Proof sketch.

1. Recursively prove arg_i = val_i for all i.
2. (fn arg_1 ... arg_n) = (fn val_1 ... val_n) by the derived rule “pequal by args”
3. (fn val_1 ... val_n) = result by base evaluation
4. Transitivity of = finishes the proof

Defined functions

Goal: (fn arg_1 ... arg_n) = result

Proof sketch.

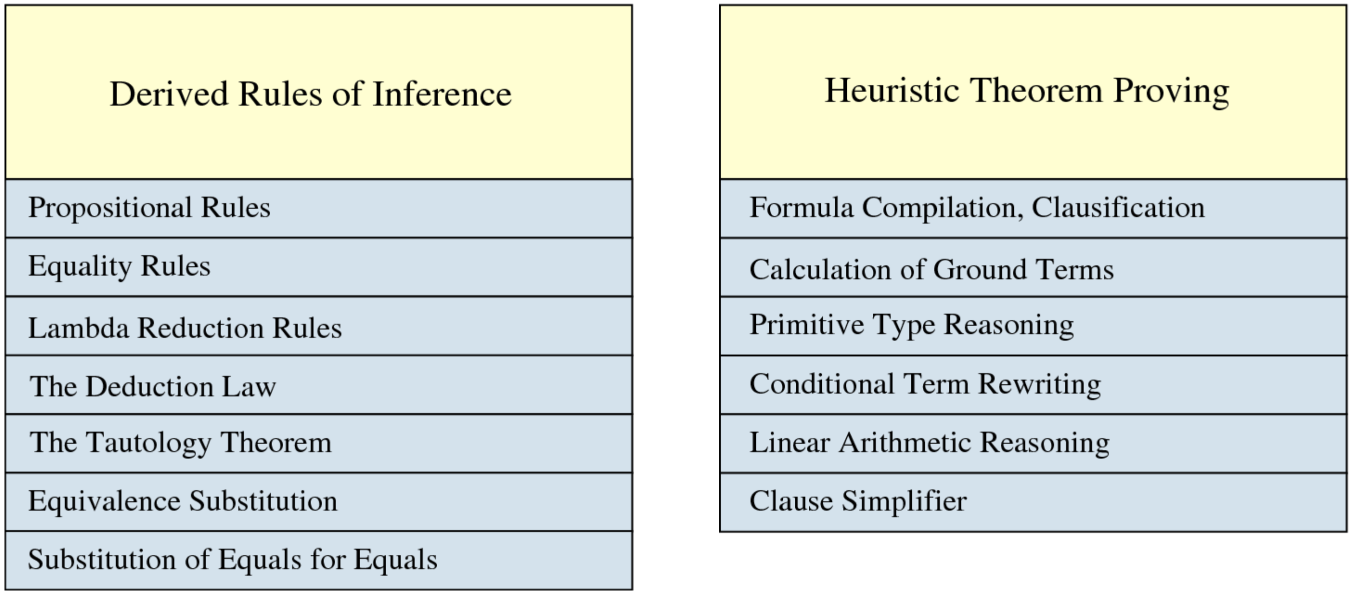
1. Recursively prove arg_i = val_i for all i
2. (fn arg_1 ... arg_n) = (fn val_1 ... val_n) by the derived rule “pequal by args”
3. (fn $formal_1$ ... $formal_n$) = body, by the definition of fn
4. (fn val_1 ... val_n) = body/[formals<-vals], by instantiation
5. Recursively prove body/[formals<-vals] = result
6. Transitivity of = finishes the proof

7. Bootstrapping

- Goal: verify extensions using proofp
  - Plan: “formalize” ACL2 proofs, which use rewriting, evaluation, etc.
  - Generic-evaluator-bldr lets us build a formal proof of a computation.
    - Example: we can prove (fact 10) = 3,628,800
- Proof size matters when bootstrapping
  - Original proof of (fib 2) = 2 was 790m conses
  - After some work it's only about 35k.
  - About 24m conses for (fib 15) now.

Beyond computation

- Evaluation can be used in new extensions:
  - Unconditional rewriter which can evaluate ground terms and apply simple rewrite rules (finished and proven sound in ACL2)
  - Conditional rewriter for a clause-based simplifier (in progress)



Derived Rules of Inference
Propositional Rules
Equality Rules
Lambda Reduction Rules
The Deduction Law
The Tautology Theorem
Equivalence Substitution
Substitution of Equals for Equals

Heuristic Theorem Proving
Formula Compilation, Clausification
Calculation of Ground Terms
Primitive Type Reasoning
Conditional Term Rewriting
Linear Arithmetic Reasoning
Clause Simplifier