Hygienic Macros for ACL2

Carl Eastlund   Matthias Felleisen

cce@ccs.neu.edu   matthias@ccs.neu.edu

Northeastern University

Boston, MA, USA

ACL2

ACL2

A COMPUTATIONAL LOGIC
ACL2
APPLICATIVE COMMON LISP

Formal verification based on pure, first-order Common Lisp.

Used to model critical hardware and software artifacts.

IBM
Rockwell Collins
Sun
ORACLE
galois

ACL2

ACL2 makes heavy use of language extensions based on unhygienic macros.

A COMPUTATIONAL LOGIC
ACL2
APPLICATIVE COMMON LISP

IBM
Rockwell Collins
Sun
ORACLE
galois

Unhygienic macros are difficult to maintain. ACL2's clients need maintainable language extensions.

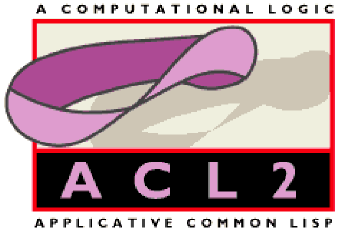
ACL2

Hygienic macros, developed for Scheme, reduce the most common pitfalls of macros.

We adapt hygienic macros to ACL2. We provide a design, prototype, and evaluation of Hygienic ACL2.

ACL2

- Computer-Aided Reasoning, Kaufmann et al., 2000
- A Computational Logic, Moore, 1979



- Hygienic Macro Expansion, Kohlbecker et al., Lisp ’86
- Macros That Work, Clinger and Rees, POPL ’91
- Syntactic Abstraction in Scheme, Dybvig et al., Lisp ’92

ACL2

```
(defun double (x) (+ x x))


(defun map-double (lst)
  (if (endp lst)
    lst
    (cons (double (car lst))
          (map-double (cdr lst)))))


(defthm len-double
  (equal (len (map-double lst))
         (len lst)))
```

ACL2

```
; Another function...
(defun square (x) (* x x))
```

ACL2

```
; Another function...
(defun square (x) (* x x))

; ...means another map.
(defun map-square (lst)
  (if (endp lst)
    lst
    (cons (square (car lst))
          (map-square (cdr lst)))))
```

ACL2

```
; Another function...
(defun square (x) (* x x))

; ...means another map.
(defun map-square (lst)
  (if (endp lst)
    lst
    (cons (square (car lst))
          (map-square (cdr lst)))))

; ACL2 is only first order!
(defthm len-square
  (equal (len (map-square lst))
         (len lst)))
```

ACL2 Macros

```
; Abstract over names...
(defmacro defun-map (map fun)
  `(defun ,map (lst)
     (if (endp lst)
       lst
       (cons (,fun (car lst))
             (,map (cdr lst))))))

; ...to generate map.
(defun-map map-double double)
```

ACL2 Macros

```
; Abstract over names...
(defmacro defun-map (map fun)
  `(defun ,map (lst)
     (if (endp lst)
       lst
       (cons (,fun (car lst))
             (,map (cdr lst))))))

; ...to generate map.
(defun map-double (lst)
  (if (endp lst)
    lst
    (cons (double (car lst))
          (map-double (cdr lst)))))
```

ACL2 Macros

```
(defmacro disprove (name body)
  `(defthm ,name (not ,body)))

(defmacro subst (e v x)
  `(let ((,x ,v)) ,e))

(defmacro top-down (top bottom)
  `(progn ,bottom ,top))

(defmacro or (a b)
  `(let ((x ,a)) (if x x ,b)))
```

ACL2 Macros

```
(defmacro or (a b)
  `(let ((x ,a)) (if x x ,b)))

(defthm excluded-middle
  (or (not x) x))
```

ACL2 Macros

```
(defmacro or (a b)
  `(let ((x ,a)) (if x x ,b)))

(defthm excluded-middle
  (let ((x (not x))) (if x x x)))
```

ACL2 Macros

```
(defmacro or (a b)
  `(let ((x ,a)) (if x x ,b)))

(defthm excluded-middle
  (let ((x (not x))) (if x x x)))

(defmacro add (a b)
  `(+ ,a ,b))

(flet ((+ (x y) (string-append x y)))
  (list (+ "thirty" "two") (add 30 2)))
```

ACL2 Macros

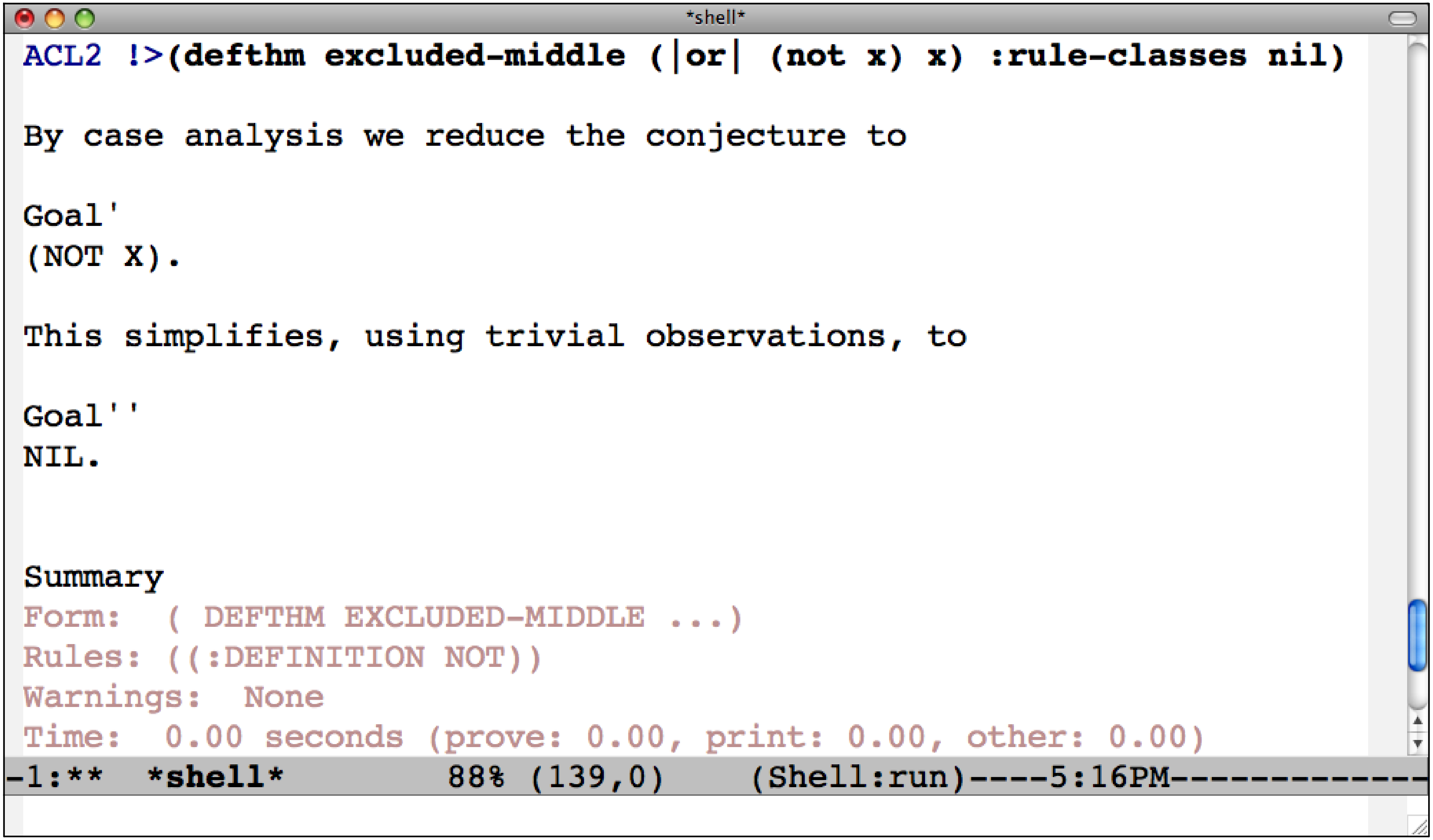
```
(defmacro or (a b)
  `(let ((x ,a)) (if x x ,b)))

(defthm excluded-middle
  (let ((x (not x))) (if x x x)))

(defmacro add (a b)
  `(+ ,a ,b))

(flet ((+ (x y) (string-append x y)))
  (list (+ "thirty" "two") (+ 30 2)))
```

ACL2 Macros



```
ACL2 !>(defthm excluded-middle (|or| (not x) x) :rule-classes nil)

By case analysis we reduce the conjecture to

Goal'
(NOT X).

This simplifies, using trivial observations, to

Goal''
NIL.


Summary
Form:  ( DEFTHM EXCLUDED-MIDDLE ...)
Rules: ((:DEFINITION NOT))
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
-1:**  *shell*        88% (139,0)     (Shell:run)----5:16PM------------
```

ACL2 Macros

```
(defstructure point x y)
```

ACL2 Macros

```
(structures::capsule
 (local (in-theory (theory 'structures::minimal-theory-for-defstructure)))
 (defun point (x y)
   (let ((point 'point))
        (cons point (cons x (cons y nil)))))
 (defthm defs-acl2-count-point
   (equal (acl2-count (point x y))
          (+ 3 (acl2-count x) (acl2-count y))))
 (defun weak-point-p (point)
   (and (consp point)
        (consp (cdr point))
        (consp (cdr (cdr point)))
        (null (cdr (cdr (cdr point))))
        (eq (car point) 'point)))
 (defthm
   defs-weak-point-p-point
   (equal (weak-point-p (point x y)) t)
   :rule-classes ((:rewrite)
                  (:built-in-clause :corollary (weak-point-p (point x y)))))
 (defun point-x (point)
   (car (cdr point)))
 (defun point-y (point)
   (car (cdr (cdr point))))
 (defun point-p (point)
   (and (weak-point-p point) t))
 (defthm defs-point-p-includes-weak-point-p
   (implies (point-p point)
            (weak-point-p point))
   :rule-classes (:forward-chaining :rewrite :built-in-clause))
 (defthm defs-point-p-point
   (equal (point-p (point x y)) t))
 (defmacro make-point
   (&whole structures::form &rest args)
   (structures::keyword-constructor-fn structures::form args 'point
                                       'make-point
                                       '((:x) (:y))
                                       '(:x :y)
                                       '(:x :y)))
```

```
(defmacro update-point
  (&whole structures::form
          structures::struct &rest args)
  (structures::keyword-updater-fn structures::form
                                  structures::struct args 'point
                                  'update-point
                                  '(:x :y)
                                  'nil
                                  ':copy
                                  '(point x y)
                                  '((:x . point-x) (:y . point-y))
                                  '((:x) (:y))))
(defthm defs-read-point
  (and (equal (point-x (point x y)) x)
       (equal (point-y (point x y)) y)))
(defthm defs-point-lift-if
  (and (equal (point-x (if point-test point-left point-right))
              (if point-test (point-x point-left)
                  (point-x point-right)))
       (equal (point-y (if point-test point-left point-right))
              (if point-test (point-y point-left)
                  (point-y point-right)))))
(defthm defs-eliminate-point
  (implies (weak-point-p point)
           (equal (point (point-x point) (point-y point))
                  point))
  :rule-classes (:rewrite :elim))
(deftheory defs-point-definition-theory
  '(point weak-point-p point-p point-x point-y))
(in-theory (disable defs-point-definition-theory))
(structures::capsule
 (deftheory defs-point-lemma-theory
   '(defs-acl2-count-point defs-eliminate-point
      defs-point-lift-if defs-point-p-point
      defs-point-p-includes-weak-point-p
      defs-read-point
      defs-weak-point-p-point)))
```

ACL2 Macros

```
(defmacro or (a b)
  (let ((x (gensym)))
    `(let ((,x ,a)) (if ,x ,x ,b))))
```

ACL2 Macros

```
~~(defmacro or (a b)
  (let ((x (gensym)))
    `(let ((,x ,a)) (if ,x ,x ,b))))~~
```

ACL2 Macros

```
(defmacro or (a b)
  `(if ,a ,a ,b))
```

ACL2 Macros

```
(defmacro or (a b)
  `(if ,a ,a ,b))

(defmacro or (a b)
  `(let ((!!!obscure ,a))
     (if !!!obscure !!!obscure ,b)))
```

ACL2 Macros

```
(defmacro or (a b)
  `(if ,a ,a ,b))

(defmacro or (a b)
  `(let ((!!!obscure ,a))
     (if !!!obscure !!!obscure ,b)))

(defmacro or (a b)
  `(let ((x ,a))
     (if x x (check-vars-not-free (x) ,b))))
```

ACL2 Macros

```
(defmacro or (a b)
  Compiler Magic!)

(defthm excluded-middle
  (or (not x) x))
```

ACL2 Macros

```
(defmacro or (a b)
  Compiler Magic!)

(defthm excluded-middle
  (or (not x) x))

(defthm excluded-middle
  (if (not x) (not x) x))
```

ACL2 Macros

```
(defmacro or (a b)
  Compiler Magic!)

(defthm excluded-middle
  (or (not x) x))

(defthm excluded-middle
  (if (not x) (not x) x))

(defthm excluded-middle
  (let ((g492 (not x))) (if g492 g492 x)))
```

Hygienic ACL2

Hygienic ACL2

Design	Policy for scope of hygienic macros.
Model	Semantics of policy-enforcing macro expander.
Prototype	Implementation as external preprocessor.
Evaluation	Comprehensive inspection of ACL2 macros.

Hygienic ACL2 Macros

```
(defmacro or (a b)
  `(let ((x ,a)) (if x x ,b)))

(defthm excluded-middle
  (or (not x) x))
```

Hygienic ACL2 Macros

```
(defmacro or (a b)
  `(let ((x ,a)) (if x x ,b)))

(defthm excluded-middle
  (let ((x (not x))) (if x x x)))
```

Hygienic ACL2 Macros

```
(defmacro or (a b)
  `(let ((x ,a)) (if x x ,b)))

(defthm excluded-middle
  (let ((x (not x))) (if x x x)))

(defmacro add (a b)
  `(+ ,a ,b))

(flet ((+ (x y) (string-append x y)))
  (list (+ "thirty" "two") (add 30 2)))
```

Hygienic ACL2 Macros

```
(defmacro or (a b)
  `(let ((x ,a)) (if x x ,b)))

(defthm excluded-middle
  (let ((x (not x))) (if x x x)))

(defmacro add (a b)
  `(+ ,a ,b))

(flet ((+ (x y) (string-append x y)))
  (list (+ "thirty" "two") (+ 30 2)))
```

Hygienic ACL2 Macros

```
(defun do-compose (funs arg)
  (if (endp funs)
    arg
    `(,(car funs) (compose ,(cdr funs) ,arg))))

(defmacro compose (funs arg)
  (do-compose funs arg))

(compose (length reverse) lst)
```

Hygienic ACL2 Macros

```
(defun do-compose (funs arg)
  (if (endp funs)
    arg
    `(,(car funs) (compose ,(cdr funs) ,arg))))

(defmacro compose (funs arg)
  (do-compose funs arg))

(length (compose (reverse) lst))
```

Hygienic ACL2 Macros

```
(defun do-compose (funs arg)
  (if (endp funs)
    arg
    `(,(car funs) (compose ,(cdr funs) ,arg))))

(defmacro compose (funs arg)
  (do-compose funs arg))

(length (reverse (compose () lst)))
```

Hygienic ACL2 Macros

```
(defun do-compose (funs arg)
  (if (endp funs)
    arg
    `(,(car funs) (compose ,(cdr funs) ,arg))))

(defmacro compose (funs arg)
  (do-compose funs arg))

(length (reverse lst))
```

Hygienic ACL2 Macros

```
(defmacro for-all (vars claim) ...)
```

```
(for-all (x y) (= (+ x y) (+ y x)))
```

Hygienic ACL2 Macros

```
(defmacro for-all (vars claim)
  `(progn
    (defun for-all-fun (,@vars) ,claim)
    (defthm for-all-thm (for-all-fun ,@vars))))

(for-all (x y) (= (+ x y) (+ y x)))
```

Hygienic ACL2 Macros

```
(defmacro for-all (vars claim)
  `(progn
    (defun for-all-fun (,@vars) ,claim)
    (defthm for-all-thm (for-all-fun ,@vars))))

(progn
 (defun for-all-fun (x y) (= (+ x y) (+ y x)))
 (defthm for-all-thm (for-all-fun x y)))
```

Hygienic ACL2 Macros

```
(defmacro for-all (vars claim)
  `(progn
    (defun for-all-fun (,@vars) ,claim)
    (defthm for-all-thm (for-all-fun ,@vars))))

(progn
 (defun for-all-fun (x y) (= (+ x y) (+ y x)))
 (defthm for-all-thm (for-all-fun x y)))

(for-all (x y) (= (* x y) (* y x)))
```

Hygienic ACL2 Macros

```
(defmacro for-all (vars claim)
  `(progn
    (defun for-all-fun (,@vars) ,claim)
    (defthm for-all-thm (for-all-fun ,@vars))))
```

```
(progn
 (defun for-all-fun (x y) (= (+ x y) (+ y x)))
 (defthm for-all-thm (for-all-fun x y)))
```

```
(progn
 (defun for-all-fun (x y) (= (* x y) (* y x)))
 (defthm for-all-thm (for-all-fun x y)))
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(include-book "A")
(include-book "B")
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(include-book "A")
(include-book "B")
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(include-book "A")
(include-book "B")
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(defun f (x) x)
(include-book "B")
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(defun f (x) x)
(defun f (x) x)
```

Hygienic ACL2 Macros

A.lisp: `(defun f (x) x)`

B.lisp: `(defun f (x) x)`

C.lisp: `(defun f (x) x)`
~~`(defun f (x) x)`~~

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(include-book "A")
(include-book "B")
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(include-book "A")
```

C.lisp:

```
(include-book "A")
(include-book "B")
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(include-book "A")
```

C.lisp:

```
(include-book "A")
(include-book "B")
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(include-book "A")
(include-book "B")
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(defun f (x) x)
(include-book "B")
```

Hygienic ACL2 Macros

A.lisp:

```
(defun f (x) x)
```

B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(defun f (x) x)
(defun f (x) x)
```

Hygienic ACL2 Macros

A.lisp:

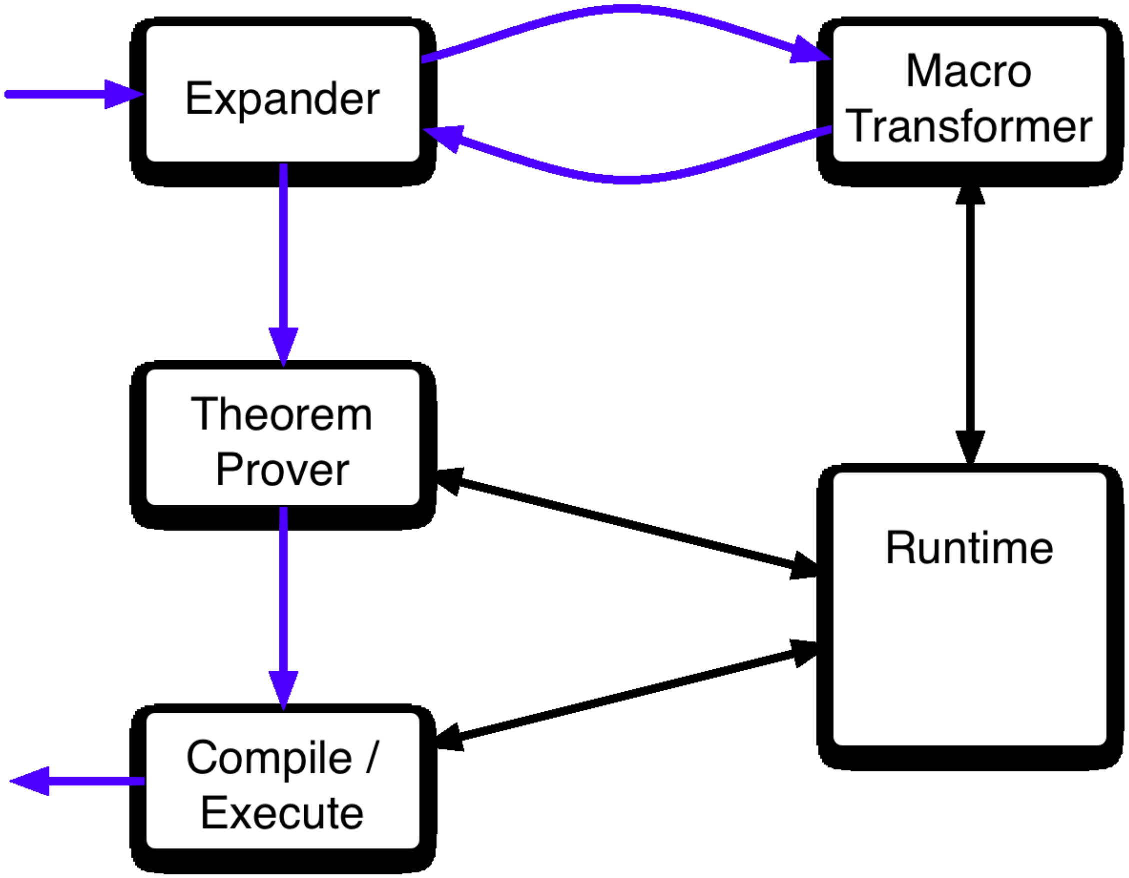
```
(defun f (x) x)
```

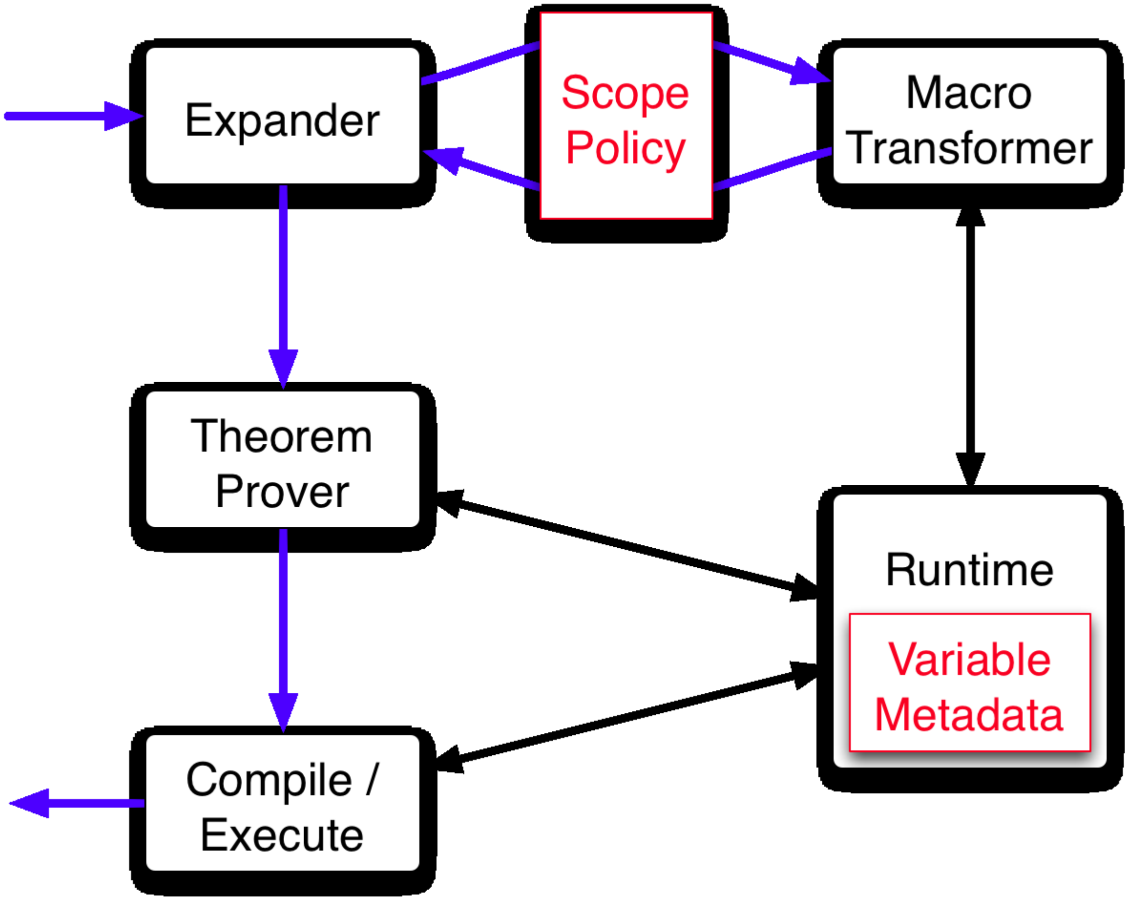
B.lisp:

```
(defun f (x) x)
```

C.lisp:

```
(defun f (x) x)
```


Expander
Macro Transformer
Theorem Prover
Runtime
Compile / Execute


Expander
Scope Policy
Macro Transformer
Theorem Prover
Runtime
Variable Metadata
Compile / Execute

Hygienic ACL2 Prototype

Implementation External preprocessor written in Scheme.

Finished Core functions and binding forms.

To Do Many extra options, functions, derived forms, and logical forms.

To Design **`state`**, **`defstobj`**, **`make-event`**

Hygienic ACL2 Evaluation

	Auto Improve	Can Improve	Same	Must Improve	Must Fix
Alias	-	-	2464	-	-
Copy	-	151	-	-	-
Refer	2	-	-	83	5
Bind	30	48	-	11	-
Define	-	2	112	44	16
Compare	-	30	-	-	-
Total	**32**	**231**	**2578**	**138**	**21**

Hygienic Macros for ACL2:

maintainable language extensions

for the existing ACL2 code base.