Hygienic Macros for the ACL2 Theorem Prover

Carl Eastlund
cce@ccs.neu.edu

Matthias Felleisen
matthias@ccs.neu.edu

Northeastern University

Boston, MA, USA

The ACL2 Theorem Prover

```
(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)))
```

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

```
; 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)))))
```

```
; 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)))
```

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

(defun-map map-double double)
```

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

(defun-map map-double double)
```

⇓

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

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

(defun find (n lst)
  (or (nth n lst) 0))
```

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

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

(defun find (n lst)
  (or (nth n lst) 0))
          ⇓
(defun find (n lst) ; Traverse twice.
  (if (nth n lst) (nth n lst) 0))

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

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

(defun find (n lst)
  (or (nth n lst) 0))
          ⇓
(defun find (n lst) ; Traverse twice.
  (if (nth n lst) (nth n lst) 0))

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

```
(defmacro or (a b) ; Bind x.
  `(let ((x ,a)) (if x x ,b)))

(defun find (n lst)
  (or (nth n lst) 0))
```

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

```
(defmacro or (a b) ; Bind x.
  `(let ((x ,a)) (if x x ,b)))

(defun find (n lst)
  (or (nth n lst) 0))
          ⇓
(defun find (n lst) ; Traverse once.
  (let ((x (nth n lst))) (if x x 0)))

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

```
(defmacro or (a b) ; Bind x.
  `(let ((x ,a)) (if x x ,b)))

(defun find (n lst)
  (or (nth n lst) 0))
          ⇓
(defun find (n lst) ; Traverse once.
  (let ((x (nth n lst))) (if x x 0)))

(defthm excluded-middle
  (or (not x) x))
           ⇓
(defthm excluded-middle ; Name clash!
  (let ((x (not x))) (if x x x)))
```

```
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------------
```

Unhygienic macros are not abstractions.

```
*shell*
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------------
```

(defstructure point x y)

⇓

```
(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)))
```

```
(defmacro or (a b) ; Special case...
  Compiler magic!)

(defun find (n lst)
  (or (nth n lst) 0))
```

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

```
(defmacro or (a b) ; Special case...
  Compiler magic!)

(defun find (n lst)
  (or (nth n lst) 0))
          ⇓
(defun find (n lst) ; Fresh variable here...
  (let ((x.1 (nth n lst))) (if x.1 x.1 0)))

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

```
(defmacro or (a b) ; Special case...
  Compiler magic!)

(defun find (n lst)
  (or (nth n lst) 0))
        ⇓
(defun find (n lst) ; Fresh variable here...
  (let ((x.1 (nth n lst))) (if x.1 x.1 0)))

(defthm excluded-middle
  (or (not x) x))
        ⇓
(defthm excluded-middle ; ...copy code here.
  (if (not x) (not x) x))
```

```
(defmacro or (a b) ; Bind x.
  `(let ((x ,a)) (if x x ,b)))

(defun find (n lst)
  (or (nth n lst) 0))
```

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

```
(defmacro or (a b) ; Bind x.
  `(let ((x ,a)) (if x x ,b)))

(defun find (n lst)
  (or (nth n lst) 0))
          ⇓
(defun find (n lst) ; Fresh variable.
  (let ((x.1 (nth n lst))) (if x.1 x.1 0)))

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

```
(defmacro or (a b) ; Bind x.
  `(let ((x ,a)) (if x x ,b)))

(defun find (n lst)
  (or (nth n lst) 0))
          ⇓
(defun find (n lst) ; Fresh variable.
  (let ((x.1 (nth n lst))) (if x.1 x.1 0)))

(defthm excluded-middle
  (or (not x) x))
          ⇓
(defthm excluded-middle ; Fresh variable.
  (let ((x.1 (not x))) (if x.1 x.1 x)))
```

Hygienic Macros

```
(define-syntax or ; Hygienic macro in Scheme.
  (syntax-rules ()
    ((or a b) (let ((x a)) (if x x b)))))

(or (not x) x)
```

```
(define-syntax or ; Hygienic macro in Scheme.
  (syntax-rules ()
    ((or a b) (let ((x a)) (if x x b)))))

(or (not x) x)
      ⇓
(or:0 (not:0 x:0) x:0)
```

```
(define-syntax or ; Hygienic macro in Scheme.
  (syntax-rules ()
    ((or a b) (let ((x a)) (if x x b)))))

(or (not x) x)
      ⇓
(or:0 (not:0 x:0) x:0)
          ⇓
(let:1 ((x:1 (not:0 x:0))) (if:1 x:1 x:1 x:0))
```

```
(define-syntax or ; Hygienic macro in Scheme.
  (syntax-rules ()
    ((or a b) (let ((x a)) (if x x b)))))

(or (not x) x)
      ⇓
(or:0 (not:0 x:0) x:0)
              ⇓
(let:1 ((x:1 (not:0 x:0))) (if:1 x:1 x:1 x:0))
                                  ⇓
(let ((x.1 (not x))) (if x.1 x.1 x))
```

```
(define-syntax or ; Hygienic macro in Scheme.
  (syntax-rules ()
    ((or a b) (let ((x a)) (if x x b)))))
```

```
(or (not x) x)
      ⇓
(or:0 (not:0 x:0) x:0)
          ⇓
(let:1 ((x:1 (not:0 x:0))) (if:1 x:1 x:1 x:0))
                                  ⇓
(let ((x.1 (not x))) (if x.1 x.1 x))
```

Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in Scheme. Lisp and Symbolic Computation 5(4) (1992) 295–326

```
; Preserve definitions.
(defmacro or (a b)
  `(let ((x ,a))
     (if x x ,b)))
```

```
; New syntax and data.
(define-syntax or
  (syntax-rules ()
    ((or a b)
     (let ((x a))
       (if x x b)))))
```

```
; Preserve definitions.
(defmacro or (a b)
  `(let ((x ,a))
     (if x x ,b)))
```

```
; New syntax and data.
(define-syntax or
  (syntax-rules ()
    ((or a b)
     (let ((x a))
       (if x x b))))))
```

```
; Preserve definitions.
(defmacro or (a b)
  `(let ((x ,a))
     (if x x ,b)))
```

```
; New syntax and data.
(define-syntax or
  (syntax-rules ()
    ((or a b)
     (let ((x a))
       (if x x b)))))
```

```
; Preserve expansion.
(defthm excluded-middle
  (let ((x (not x)))
    (if x x x)))
```

```
; Hygienic expansion.
(defthm excluded-middle
  (let ((x.1 (not x)))
    (if x.1 x.1 x)))
```

```
; Preserve definitions.
(defmacro or (a b)
  `(let ((x ,a))
     (if x x ,b)))
```

```
; New syntax and data.
(define-syntax or
  (syntax-rules ()
    ((or a b)
     (let ((x a))
       (if x x b)))))
```

```
; Preserve expansion.
(defthm excluded-middle
  (let ((x (not x)))
    (if x x x)))
```

```
; Hygienic expansion.
(defthm excluded-middle
  (let ((x.1 (not x)))
    (if x.1 x.1 x)))
```

```
; Preserve definitions.
(defmacro or (a b)
  `(let ((x ,a))
     (if x x ,b)))
```

```
; New syntax and data.
(define-syntax or
  (syntax-rules ()
    ((or a b)
     (let ((x a))
       (if x x b)))))
```

```
; Preserve expansion.
(defthm excluded-middle
  (let ((x (not x)))
    (if x x x)))
```

```
; Hygienic expansion.
(defthm excluded-middle
  (let ((x.1 (not x)))
    (if x.1 x.1 x)))
```

```
; Preserve axioms.
(defthm x=x
  (equal x:0 x:1))
```

```
; Model hygiene in ACL2.
(defthm x!=x
  (not (equal x:0 x:1)))
```

```
; Preserve definitions.
(defmacro or (a b)
  `(let ((x ,a))
     (if x x ,b)))
```

```
; New syntax and data.
(define-syntax or
  (syntax-rules ()
    ((or a b)
     (let ((x a))
       (if x x b)))))
```

```
; Preserve expansion.
(defthm excluded-middle
  (let ((x (not x)))
    (if x x x)))
```

```
; Hygienic expansion.
(defthm excluded-middle
  (let ((x.1 (not x)))
    (if x.1 x.1 x)))
```

```
; Preserve axioms.
(defthm x=x
  (equal x:0 x:1))
```

```
; Model hygiene in ACL2.
(defthm x!=x
  (not (equal x:0 x:1)))
```

```
; Preserve definitions.
(defmacro or (a b)
  `(let ((x ,a))
     (if x x ,b)))
```

```
; Hygienic expansion.
(defthm excluded-middle
  (let ((x.1 (not x)))
    (if x.1 x.1 x)))
```

```
; Preserve axioms.
(defthm x=x
  (equal x:0 x:1))
```

Evolving Hygiene

```
(defun not-not (x)
  (let ((not (not x)))
    (not not)))
```

```
(defun not-not (x)
  (let ((not (not x)))
    (not not)))
```

⇓

```
(defun not-not (x)
  (let ((not.1 (not x)))
    ; Mis-renamed function call.
    (not.1 not.1)))
```

```
(defun not-not (x)
  (let ((not (not x)))
    (not not)))
```

⇓

```
(defun not-not (x)
  (let ((not.1 (not x)))
    ; Rename only local variable.
    (not not.1)))
```

```
(let ((init 0))
  (let ((result init))
    result))

; init and result are free here:
(list init result)
```

```
(let ((init 0))
  (let ((result init))
    result))

; init and result are free here:
(list init result)
```

⇓

```
(let ((init.1 0))
  (let ((result.1 init.1))
    result.1))

(list init result)
```

```
(encapsulate () ; exports main and action
  (encapsulate () ; exports action, hides helper
    (local (defun helper (x) x))
    (defun action (x) (helper x)))
  (defun main (x) (action x)))

; main and action are bound here:
(main (action (helper 0)))
```

```
(encapsulate () ; exports main and action
  (encapsulate () ; exports action, hides helper
    (local (defun helper (x) x))
    (defun action (x) (helper x)))
  (defun main (x) (action x)))

; main and action are bound here:
(main (action (helper 0)))
```

⇓

```
(encapsulate ()
  (encapsulate ()
    (local (defun helper.1 (x) x))
    (defun action.1 (x) (helper.1 x)))
  (defun main.1 (x) (action.1 x)))

(main.1 (action.1 (helper 0)))
```

```
(defmacro defun-map (map fun)

    `(defun ,map (lst)
       (if (endp lst)
           lst
           (cons (,fun (car lst))
                 (,map (cdr lst))))))

(defun-map map-double double)
```

```lisp
; Construct map-<fun> implicitly.
(defmacro defun-map (fun)
  (let ((map (intern (prefix "map-" fun) "ACL2")))
    `(defun ,map (lst)
       (if (endp lst)
           lst
           (cons (,fun (car lst))
                 (,map (cdr lst)))))))

(defun-map double)
```

```lisp
; Construct map-<fun> implicitly.
(defmacro defun-map (fun)
  (let ((map (intern (prefix "map-" fun) "ACL2")))
    `(defun ,map (lst)
       (if (endp lst)
           lst
           (cons (,fun (car lst))
                 (,map (cdr lst)))))))

(defun-map double)
                 ⇓
; Unhygienic expansion.
(defun map-double (lst)
  (if (endp lst)
      lst
      (cons (double (car lst))
            (map-double (cdr lst)))))
```

```
; Construct map-<fun> implicitly.
(defmacro defun-map (fun)
  (let ((map (intern (prefix "map-" fun) "ACL2")))
    `(defun ,map (lst)
       (if (endp lst)
           lst
           (cons (,fun (car lst))
                 (,map (cdr lst)))))))

(defun-map double)
                ⇓
; Oops.  Bound in wrong context.
(defun map-double.1 (lst.1)
  (if (endp lst.1)
      lst.1
      (cons (double (car lst.1))
            (map-double.1 (cdr lst.1)))))
```

```
; Copy hygiene info to map-<fun>.
(defmacro defun-map (fun)
  (let ((map (i-p-s (prefix "map-" fun) fun)))
    `(defun ,map (lst)
       (if (endp lst)
           lst
           (cons (,fun (car lst))
                 (,map (cdr lst)))))))

(defun-map double)
                 ⇓
; Correct context.
(defun map-double (lst.1)
  (if (endp lst.1)
      lst.1
      (cons (double (car lst.1))
            (map-double (cdr lst.1)))))
```

The ACL2 theorem prover has over 1,000,000 lines of libraries and regression tests.

Who knows what idioms their macros may include?

Hygienic ACL2:

hygienic,

logically sound,

backwards compatible

macro system

for the ACL2 theorem prover.

Hygienic ACL2:

hygienic,

logically sound,

backwards compatible

macro system

for the ACL2 theorem prover.

http://www.ccs.neu.edu/~cce/acl2