Hygienic Macros for the ACL2 Theorem Prover

Carl Eastlund  Matthias Felleisen

cce@ccs.neu.edu  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))

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

(defun 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)
Unhygienic macros are not abstractions.

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)
(defstructure point x y)

(structures : capsule
  (local (in-theory (theory 'structures : minimal-theory-for-defstructure))
    (defun point (x y)
      (list (point 'point)
        (cons point (cons x (cons y nil)))))
    (defthm defACL2-count-point
      (equal (ACL2-COUNT (point x y))
        (+ 3 (ACL2-COUNT X) (ACL2-COUNT Y))))
    (defun near-point-p (point)
      (and (consp point)
        (consp (car point))
        (cons (consp (cdr point))
          (null (consp (consp (cdr point))))))
      (eq (car point) 'point)))
    (defthm defNear-point-p-point
      (equal (near-point-p (point x y)) t)
      :rule-classes (:rewrite)
      :built-in-clause :corollary (near-point-p (point x y)))
    (defun point-x (point)
      (car (car point)))
    (defun point-y (point)
      (car (cdr point)))
    (defun point-p (point)
      (and (near-point-p point)
        (near-point-p point) t)
      :rule-classes (:forward-chain :remote :built-in-clause)
      :defthm defNear-point-p-point
      (equal (point-p (point x y)) t))
    (defmacro make-point
      (args)
      (list (make-point (point 'point)
        (cons (x) (y))))
      :rule-updater-fn (defstructures::keyword-constructor-fn)
      &whole)
    (defthm defNear-point-lemma-theory
      "near-point-lemma"
      (implies (near-point-p point)
        (equal (point-x point) (x))
        (equal (point-y point) (y)))
      :rule-classes (:(remote :else))
      :disable (defstructures::keyword-updater-fn)
      (defstructures::struct-rewrite-point)
      (defstructures::struct-rewrite-nil)
      (defstructures::minimal-theory-for-defstructure)
      (defstructures::form-nil)
      (defstructures::form-nil :for-forward-chaining)
      (defstructures::form-nil :for-rewrite)
      (defstructures::minimal-theory-nil)
      :theory (defstructures::minimal-theory-nil))
    (defthm defNear-point-definition-theory
      (implies (near-point-x (point x y)) (x)
        (equal (point-x (point x y)) (x)))
      :rule-classes (:(remote :else))
      :disable (defstructures::keyword-updater-fn)
      (defstructures::struct-rewrite-point)
      (defstructures::struct-rewrite-nil)
      (defstructures::minimal-theory-nil)
      :theory (defstructures::minimal-theory-nil))
    (defthm defNear-point-elimination-theory
      (implies (near-point-x (point x y)) (x)
        (equal (point-x (point x y)) (x)))
      :rule-classes (:(remote :else))
      :disable (defstructures::keyword-updater-fn)
      (defstructures::struct-rewrite-point)
      (defstructures::struct-rewrite-nil)
      (defstructures::minimal-theory-nil)
      :theory (defstructures::minimal-theory-nil))
    (defthm defNear-point-definition-nil
      (implies (near-point-x (point x y)) (x)
        (equal (point-x (point x y)) (x)))
      :rule-classes (:(remote :else))
      :disable (defstructures::keyword-updater-fn)
      (defstructures::struct-rewrite-point)
      (defstructures::struct-rewrite-nil)
      (defstructures::minimal-theory-nil)
      :theory (defstructures::minimal-theory-nil))
    (defthm defNear-point-elimination-nil
      (implies (near-point-x (point x y)) (x)
        (equal (point-x (point x y)) (x)))
      :rule-classes (:(remote :else))
      :disable (defstructures::keyword-updater-fn)
      (defstructures::struct-rewrite-point)
      (defstructures::struct-rewrite-nil)
      (defstructures::minimal-theory-nil)
      :theory (defstructures::minimal-theory-nil))
    (defthm defNear-point-nil-nil
      (implies (near-point-x (point x y)) (x)
        (equal (point-x (point x y)) (x)))
      :rule-classes (:(remote :else))
      :disable (defstructures::keyword-updater-fn)
      (defstructures::struct-rewrite-point)
      (defstructures::struct-rewrite-nil)
      (defstructures::minimal-theory-nil)
      :theory (defstructures::minimal-theory-nil)))

(structures : minimal-theory-nil)

(deftheory defstructures-nil-definition-theory
  :theory (defstructures::minimal-theory-nil))

(structures : capsule
  (defstructures::minimal-theory-nil)
  (defstructures::form-nil)
  (defstructures::form-nil :for-forward-chaining)
  (defstructures::form-nil :for-rewrite)
  (defstructures::minimal-theory-nil)
  :theory (defstructures::minimal-theory-nil)))

17
(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)))

(defun 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-syntex 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)
\[
\begin{align*}
  \Downarrow \\
  (or:0 (not:0 x:0) x:0) \\
  \Downarrow \\
  (let:1 ((x:1 (not:0 x:0))) (if:1 x:1 x:1 x:0)) \\
  \Downarrow \\
  (let ((x.1 (not x))) (if x.1 x.1 x))
\end{align*}
\]
(define-syntax or ; Hygienic macro in Scheme.
  (syntax-rules ()
    ((or a b) (let ((x a)) (if x x b))))

(or (not x) x)
  \[\Rightarrow\]
(or:0 (not:0 x:0) x:0)
    \[\Rightarrow\]
(let:1 ((x:1 (not:0 x:0))) (if:1 x:1 x:1 x:0))
      \[\Rightarrow\]
(let ((x.1 (not x))) (if x.1 x.1 x))

; Preserve definitions. ; New syntax and data.
(defmacro or (a b)
  `(let ((x ,a))
     (if x x ,b)))

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

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

; Preserve expansion.  ; Hygienic expansion.
(defthm excluded-middle (let ((x (not x)))
    (if x x x))
(defthm excluded-middle (let ((x.1 (not x)))
    (if x.1 x.1 x)))
; Preserve definitions.  ; New syntax and data.
(defmacro or (a b)  (define-syntax or
  `(let ((x ,a))  (syntax-rules ()
    ((or a b)  ((or a b)
      (let ((x a))  (let ((x a))
        (if x x b)))))))
  (if x x ,b)))

; Preserve expansion.  ; Hygienic expansion.
(defthm excluded-middle  (defthm excluded-middle
  (let ((x (not x)))  (let ((x.1 (not x)))
    (if x x x)))  (if x.1 x.1 x)))
\[
\text{; Preserve definitions.}
\]
(defmacro or (a b)
  `(let ((x ,a))
      (if x x ,b)))

\[
\text{; New syntax and data.}
\]
(define-syntax or
  (syntax-rules ()
    ((or a b)
      (let ((x a))
        (if x x b)))))

\[
\text{; Preserve expansion.}
\]
(defthm excluded-middle
  (let ((x (not x)))
    (if x x x)))

\[
\text{; Hygienic expansion.}
\]
(defthm excluded-middle
  (let ((x.1 (not x)))
    (if x.1 x.1 x)))

\[
\text{; Preserve axioms.}
\]
(defthm x=x
  (equal x:0 x:1))

\[
\text{; Model hygiene in ACL2.}
\]
(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.1)))

; 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)))))

(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)
; 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)
; 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