Ptest-forms.lisp

```
;;; For Use in demonstration on 7/20/05
;;; Also serves as a good check list for "oh yeah, this should work"

; Process outside ACL2 loop

; Can use the lexical variable inside the binding calculation
(let ((q 10)) (plet ((x 4) (y (+ 9 q))) (+ x y)))

; Can use the lexical variable inside the body of the let
(let ((q 10)) (plet ((x 4) (y 9)) (+ x y q)))

; Can use it everywhere
(let ((q 10)) (plet ((x 4) (y (+ 9 q))) (+ x y q)))

; Can use globals
(setf *tmp* 4)
(plet ((x *tmp*)) x)
```

Ptest-arithmetic.lisp

```
(defun identity-burn-x-cpu-cycles (x acc)
  (declare (xargs :guard (and (natp x)
                              (natp acc))))
  (if (zp x)
      acc
    (identity-burn-x-cpu-cycles (- x 1) (+ 1 acc))))


(defun identity-burn-x-seconds (x)
  (declare (xargs :guard (natp x)))
  (/ (identity-burn-x-cpu-cycles (* x 17695678) 0) 17695678))


(defun arithmetic (x)
  (declare (xargs :guard (natp x)))
  (+ (identity-burn-x-seconds x)
     (identity-burn-x-seconds x)))


(defun parithmetic (x)
  (declare (xargs :guard (natp x)))
  (parallelize (+ (identity-burn-x-seconds x)
                  (identity-burn-x-seconds x))))


(time$ (arithmetic 10))
(time$ (parithmetic 10))
```

Ptest-stobj.lisp

```
(defstobj foo field1 field2)

(defun test-stobj (x foo)
  (declare (xargs :stobjs foo))
  (plet ((foo (update-field1 17 foo)))
        (update-field2 x foo)))

(test-stobj 14 foo)
(field1 foo)
(field2 foo)


(defun test-stobj-read (foo)
  (declare (xargs :stobjs foo
                  :guard (and (acl2-numberp (field1 foo))
                              (acl2-numberp (field2 foo)))))
  (let ((foo1 (field1 foo))
        (foo2 (field2 foo)))
    (+ foo1 foo2)))

(test-stobj-read foo)

(defun test-stobj-pread (foo)
  (declare (xargs :stobjs foo
                  :guard (and (acl2-numberp (field1 foo))
                              (acl2-numberp (field2 foo)))))
  (plet ((foo1 (field1 foo))
         (foo2 (field2 foo)))
    (+ foo1 foo2)))

(test-stobj-pread foo)


; The following does intentionally not admit
(defun test-stobj-pwrite (foo)
  (declare (xargs :stobjs foo
                  :guard (and (acl2-numberp (field1 foo))
                              (acl2-numberp (field2 foo)))))
  (plet ((foo (update-field1 foo 14))
         (foo (update-field2 foo 15)))
        (field1 foo)))

; Ask audience if there are other permutations they'd like to try
```

```
Ptest-mergesort.lisp


;;;;;;;;;;;;;;;; Tests for presentation on 07/20/05 ;;;;;;;;;;;;;


(include-book "finite-set-theory/osets/sets" :dir :system)

(defun integers (n acc)
  (if (zp n)
      (reverse acc)
    (integers (1- n) (cons n acc))))


:redef


; Tail-recursive versions of split-list
(skip-proofs
 (defun sets::split-list-aux (x acc0 acc1)
   (declare (xargs :guard (true-listp x)))
   (cond ((endp x) (mv acc0 acc1))
         ((endp (cdr x)) (mv (cons (car x) acc0) acc1))
         (t (sets::split-list-aux (cddr x)
                                  (cons (car x) acc0)
                                  (cons (cadr x) acc1))))))

(skip-proofs
 (defun sets::split-list (x)
   (declare (xargs :guard (true-listp x)))
   (sets::split-list-aux x nil nil)))


; Non parallel version
(skip-proofs
 (defun SETS::mergesort-exec (x)
   (cond ((endp x) nil)
         ((endp (cdr x)) (SETS::insert (car x) nil))
         (t (mv-let (part1 part2)
                    (SETS::split-list x)
                    (SETS::union (SETS::mergesort-exec part1)
(SETS::mergesort-exec part2)))))))

:q
(ccl:set-lisp-heap-gc-threshold 50000000)
(ccl:egc nil)
(setf *my-ints* (integers 50000 nil))

(print-object "Non parallel version is in effect" *standard-output*)

(time (SETS::mergesort-exec *my-ints*))




; Parallel version
```

```
(skip-proofs
 (defun SETS::mergesort-exec (x)
   (cond ((endp x) nil)
         ((endp (cdr x)) (SETS::insert (car x) nil))
         (t (mv-let (part1 part2)
                    (SETS::split-list x)
                    (ACL2::parallelize (SETS::union (SETS::mergesort-
exec part1)
                                                    (SETS::mergesort-
exec part2)))))))))


(assert$ (equal (length (SETS::mergesort-exec (integers 50000 nil)))
                50000)
         "No longer dropping elements. Thanks Matt for multiple-value-
bind!")

(time (SETS::mergesort-exec *my-ints*))



; Parallel version with granularity function
; It's interesting to note that the (length x) is extremely fast
compared to
; the expensive parts of computation
(skip-proofs
 (defun SETS::mergesort-exec (x)
   (cond ((endp x) nil)
         ((endp (cdr x)) (SETS::insert (car x) nil))
         (t (mv-let (part1 part2)
                    (SETS::split-list x)
                    (ACL2::parallelize (SETS::union (SETS::mergesort-
exec part1)
                                                    (SETS::mergesort-
exec part2))
                                            (> (length x) 40000)))))))


(time (SETS::mergesort-exec *my-ints*))
```

Ptest-adaptive.lisp

```
; Identity function that burns cpu cycles and uses constant stack space
(defun identity-burn-x-cpu-cycles (x acc)
  (declare (xargs :guard (and (natp x)
                              (natp acc))))
  (if (zp x)
      acc
    (identity-burn-x-cpu-cycles (- x 1) (+ 1 acc))))


(defun identity-burn-x-seconds (x)
  (declare (xargs :guard (natp x)))
  (/ (identity-burn-x-cpu-cycles (* x 17695678) 0) 17695678))




(defun test-let (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (let ((a (identity-burn-x-seconds x))
          (b (test-let (1- x))))
      (+ a b))))


(defun test-plet-PL (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (plet ((a (test-plet-PL (- x 1)))
           (b (identity-burn-x-seconds x)))
          (+ a b))))


(defun test-plet-PR (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (plet ((a (identity-burn-x-seconds x))
           (b (test-plet-PR (- x 1))))
          (+ a b))))




(defun test-let2 (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (let ((a (identity-burn-x-seconds x))
          (b (identity-burn-x-seconds x))
          (c (test-let2 (1- x))))
      (+ a b c))))
```

```
(defun test-plet2-PL (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (plet ((a (test-plet2-PL (- x 1)))
           (b (identity-burn-x-seconds x))
           (c (identity-burn-x-seconds x)))
          (+ a b c))))


(defun test-plet2-PR (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (plet ((a (identity-burn-x-seconds x))
           (b (identity-burn-x-seconds x))
           (c (test-plet2-PR (- x 1))))
          (+ a b c))))




(defun test-let3 (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (let ((a (identity-burn-x-seconds x))
          (b (identity-burn-x-seconds x))
          (c (identity-burn-x-seconds x))
          (d (test-let3 (1- x))))
      (+ a b c d))))



(defun test-plet3-PL (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (plet ((a (test-plet3-PL (- x 1)))
           (b (identity-burn-x-seconds x))
           (c (identity-burn-x-seconds x))
           (d (identity-burn-x-seconds x)))
          (+ a b c d))))


(defun test-plet3-PR (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (plet ((a (identity-burn-x-seconds x))
           (b (identity-burn-x-seconds x))
           (c (identity-burn-x-seconds x))
           (d (test-plet3-PR (- x 1))))
          (+ a b c d))))
```

```
(old-trace test-let2)
(old-trace test-plet2-PL)
(old-trace test-plet2-PR)
(old-trace parallelize-fn)
(old-trace identity-burn-x-seconds)



(princ$ "Beginning Testing Plet 1" *standard-co* state)
(time$ (test-let 10))
(time$ (test-let 10))
(time$ (test-let 10))

(time$ (test-plet-PL 10))
(time$ (test-plet-PL 10))
(time$ (test-plet-PL 10))

(time$ (test-plet-PR 10))
(time$ (test-plet-PR 10))
(time$ (test-plet-PR 10))

(princ$ "Beginning Testing PLet 2" *standard-co* state)


(time$ (test-let2 10))
(time$ (test-let2 10))
(time$ (test-let2 10))

(time$ (test-plet2-PL 10))
(time$ (test-plet2-PL 10))
(time$ (test-plet2-PL 10))

(time$ (test-plet2-PR 10))
(time$ (test-plet2-PR 10))
(time$ (test-plet2-PR 10))

(princ$ "Beginning Testing Plet 3" *standard-co* state)

(time$ (test-let3 10))
(time$ (test-let3 10))
(time$ (test-let3 10))

(time$ (test-plet3-PL 10))
(time$ (test-plet3-PL 10))
(time$ (test-plet3-PL 10))

(time$ (test-plet3-PR 10))
(time$ (test-plet3-PR 10))
(time$ (test-plet3-PR 10))

(princ$ "Done Testing Plet" *standard-co* state)



(defun test-and2 (x)
  (declare (xargs :guard (natp x)))
```

```
  (if (zp x)
      0
    (and (identity-burn-x-seconds x)
         (identity-burn-x-seconds x)
         (test-and2 (1- x)))))


(defun test-pand2-PR (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (pand (identity-burn-x-seconds x)
          (identity-burn-x-seconds x)
          (test-pand2-PR (- x 1)))))

(defun test-or2 (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (or (identity-burn-x-seconds x)
        (identity-burn-x-seconds x)
        (test-or2 (1- x)))))

; While lazy evaluation isn't full here, at least we don't spend the
full time
; we could spend
(defun test-por2-PR (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (por (identity-burn-x-seconds x)
         (identity-burn-x-seconds x)
         (test-por2-PR (- x 1)))))


#|
(defun parallelize-fn (parent-fun-name arg-closures &optional
terminate-early-function)
  (eval (cons parent-fun-name (parallelize-closures arg-closures
terminate-early-function))))
|#

(time$ (test-and2 10))
(time$ (test-and2 10))
(time$ (test-and2 10))

(time$ (test-pand2-PR 10))
(time$ (test-pand2-PR 10))
(time$ (test-pand2-PR 10))


(time$ (test-or2 10))
(time$ (test-or2 10))
(time$ (test-or2 10))

(time$ (test-por2-PR 10))
(time$ (test-por2-PR 10))
```

```
(time$ (test-por2-PR 10))
```

Ptest-fib.lisp

```
(skip-proofs
 (defun fib (x)
   (declare (xargs :guard (natp x)
                   :measure (acl2-count x)))
   (if (<= x 1)
       1
     (let ((a (fib (- x 1)))
           (b (fib (- x 2))))
       (+ a b)))))

#|
(skip-proofs
 (defun pfib (x)
   (declare (xargs :guard (natp x)
                   :measure (acl2-count x)))
   (if (<= x 1)
       1
     (plet ((a (pfib (- x 1)))
            (b (pfib (- x 2))))
       (+ a b)))))
|#


(skip-proofs
 (defun pfib (x)
   (declare (xargs :guard (natp x)
                   :measure (acl2-count x)))
   (if (<= x 1)
       1
     (plet ((a (pfib (- x 1)))
            (b (pfib (- x 2))))
            (+ a b)
            (> x 35)))))



(princ$ "Testing Fib (40)" *standard-co* state)

(time$ (fib 40))
(time$ (fib 40))
(time$ (fib 40))

(time$ (pfib 40))
(time$ (pfib 40))
(time$ (pfib 40))

(princ$ "Testing Fib (45)" *standard-co* state)

(time$ (fib 45))
(time$ (fib 45))
(time$ (fib 45))

(time$ (pfib 45))
(time$ (pfib 45))
(time$ (pfib 45))
```

```
(princ$ "Testing Fib (47)" *standard-co* state)

(time$ (fib 47))
(time$ (fib 47))
(time$ (fib 47))

(time$ (pfib 47))
(time$ (pfib 47))
(time$ (pfib 47))

(princ$ "Done Testing Fib" *standard-co* state)
```