



# Fast Records for ACL2

Jared Davis

ACL2 Seminar, October 12, 2005

# Outline

- Kinds of Efficiency in ACL2
- What are Records?
- Rob Sumners' Records Library
- Memtrees
- Combining Memtrees with Records
- Optimizing the Combination

# Kinds of Efficiency in ACL2

- Execution Efficiency
  - How quickly do your functions execute?
  - How much memory do they use?
  - Important for running simulations, testing
- Reasoning Efficiency
  - How quickly can your rewrite rules be applied?
    - Fewer hypotheses = faster rule application
    - Fewer case splits = fewer cases to consider
  - Important for large-scale theorem proving

# What are Records?

- Bindings of values to names
  - Also called maps or finite functions
  - (get key rec) returns the binding of key in rec
  - (set key val rec) sets binds key to val in rec
- Possible implementations
  - Positional lists (nth, update-nth)
  - Alists (assoc, acons)
  - Macros like defrec, defrecord

# Rob Sumners' Records Library

- misc/records.lisp
  - Matt Kaufmann and Rob Sumners. *Efficient Rewriting of Operations on Finite Data Structures in ACL2*. ACL2 Workshop 2002.
  - (g a r) gets value of  $a$  from record  $r$
  - (s a v r) sets value of  $a$  to  $v$  in record  $r$
- Focuses on efficient reasoning

# The Record Theorems

- No type hypotheses!
  - $(g\ a\ (s\ a\ v\ r)) = v$
  - If  $a \neq b$ , then  $(g\ a\ (s\ b\ v\ r)) = (g\ a\ r)$
  - $(s\ a\ y\ (s\ a\ x\ r)) = (s\ a\ y\ r)$
  - If  $a \neq b$ , then  $(s\ b\ y\ (s\ a\ x\ r)) = (s\ a\ x\ (s\ b\ y\ r))$
  - $(s\ a\ (g\ a\ r)\ r) = r$
- All these theorems can be satisfied simultaneously?

# Implementation

- **(recordp x)** recognizes only those alists, x, s.t. the keys of x are fully ordered using <<, and the values of x do not contain nil
- **(g-aux a r)** gets the value a is bound to in the recordp r, or nil if a is not bound in r.
- **(s-aux a v r)** updates the value bound to a in the recordp r to v.

(Note: s-aux should return a valid recordp, so if v is nil, it must erase a from r.)

# Some Easy Theorems

- If (recordp r), then

- $(g\text{-aux } a (s\text{-aux } a v r)) = v$

- If  $a \neq b$ , then  $(g\text{-aux } a (s\text{-aux } b v r)) = (g\text{-aux } a r)$

- $(s\text{-aux } a y (s\text{-aux } a x r)) = (s\text{-aux } a y r)$

- If  $a \neq b$ , then

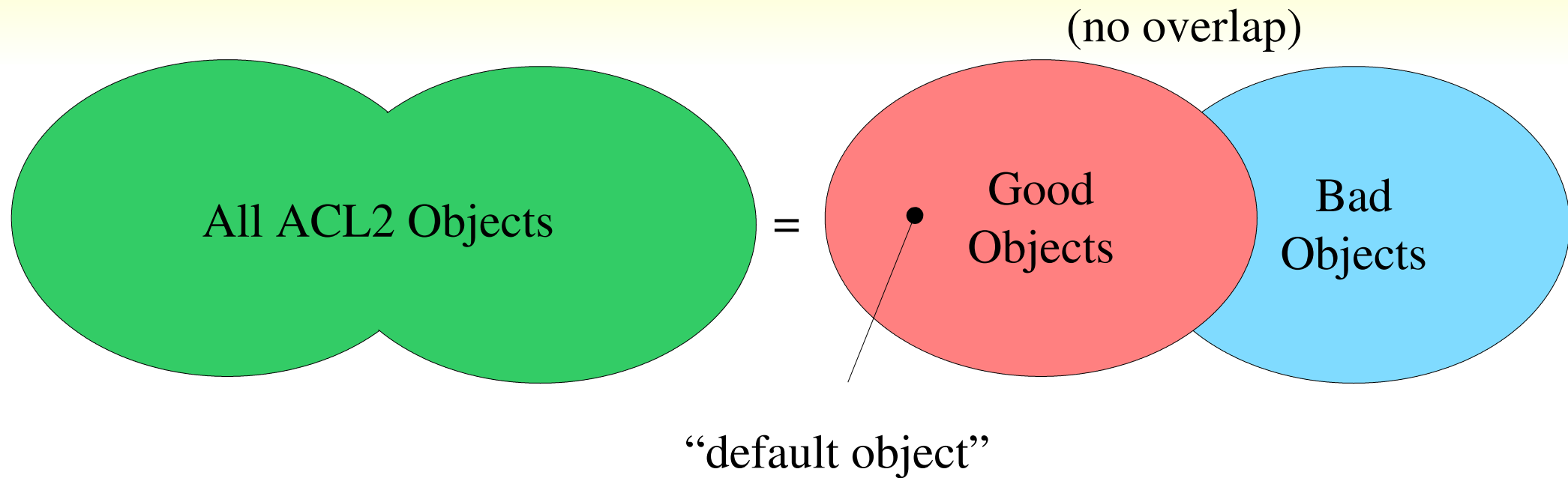
- $(s\text{-aux } b y (s\text{-aux } a x r)) = (s\text{-aux } a x (s\text{-aux } b y r))$

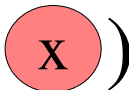
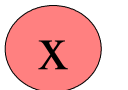
- $(s\text{-aux } a (g\text{-aux } a r) r) = r$

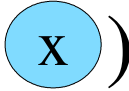
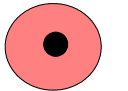
- But these still have the type hypothesis about “r”



# Fixing Functions



(Fix  ) = 

(Fix  ) = 

```
(defun foo (x y)
  (let ((x (fix x)))
    ((y (fix y))))
  ...)
```

# Fixing Functions (2)

- Fixing arguments sometimes lets us remove type hypotheses.

Examples:

$$(+ a (* b c)) = (+ (* a b) (* a c))$$

Because + and \* treat non-numeric arguments as “0”

$$(\text{subset } x \ x)$$

Because subset treats non-set arguments as “nil”

- Will this give us the record hypotheses?

# “Inadequacy” of Fixing (1)

- Proposal:
  - Let  $(g \ a \ r) = (g\text{-aux } a \ (\text{fix } r))$
  - Let  $(s \ a \ v \ r) = (s\text{-aux } a \ v \ (\text{fix } r))$
- Problem: can't prove  $(s \ a \ (g \ a \ r) \ r) = r$ 
  - $(s \ a \ (g \ a \ r) \ r) = (s\text{-aux } a \ (g \ a \ r) \ (\text{fix } r))$
  - $(\text{fix } r)$  is good, and  $s\text{-aux}$  preserves goodness
  - So  $(s \ a \ (g \ a \ r) \ r)$  is always good.
  - So any bad  $r$  is a counterexample.
- Fixing “forgets” which bad value we had.

# “Inadequacy” of Fixing (2)

- Patch Proposal to Prove  $(s\ a\ (g\ a\ r)\ r) = r$ 
  - Let  $(g\ a\ r) = (g\text{-aux}\ a\ (\text{fix}\ r))$
  - Let  $(s\ a\ v\ r) = (\text{if}\ (= (g\ a\ r)\ v)\ r\ (s\text{-aux}\ a\ v\ (\text{fix}\ r)))$
- Can't prove  $(s\ a\ x\ (s\ a\ y\ r)) = (s\ a\ x\ r)$

Suppose  $x \neq y$ ,  $(g\ a\ r) = x$ , and  $r$  is bad

$(s\ a\ x\ r) = r.$

$(g\ a\ (s\text{-aux}\ a\ y\ (\text{fix}\ r))) = y.$

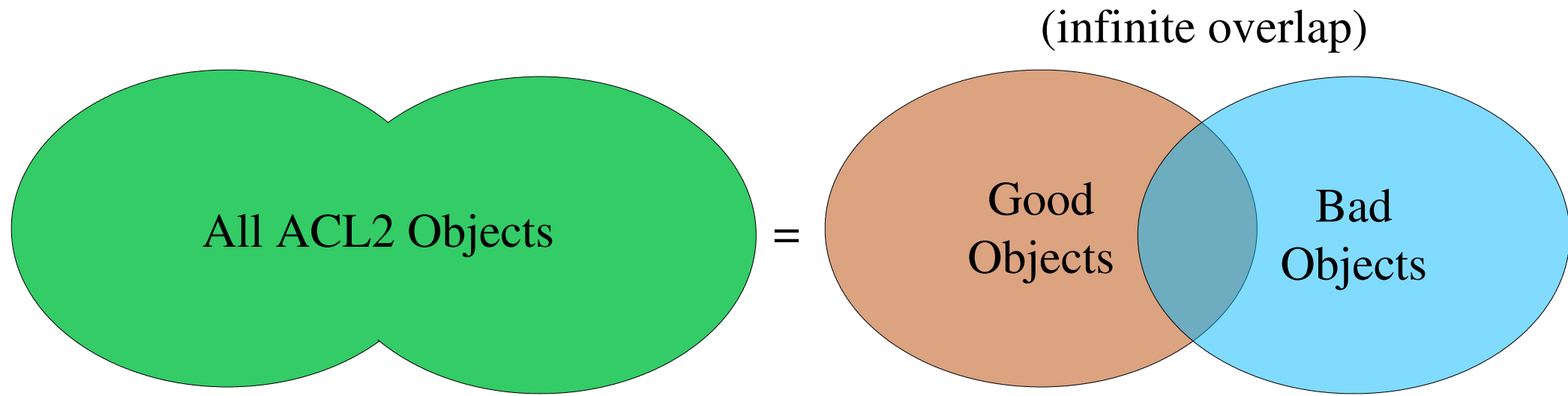
$(s\ a\ x\ (s\ a\ y\ r)) = (s\ a\ x\ (s\text{-aux}\ a\ y\ (\text{fix}\ r)))$

$= (s\text{-aux}\ a\ x\ (s\text{-aux}\ a\ y\ (\text{fix}\ r)))$

$= (s\text{-aux}\ a\ x\ (\text{fix}\ r)) \quad (\text{must be good!})$

# A Neat Trick

- **(badp x)** recognizes:
  - Any object which is not an recordp
  - Any recordp of the form ((nil . y)), where (badp y).
- Infinite overlap between our good and bad objects; infinite “default objects”

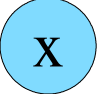
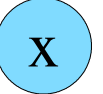





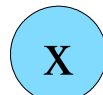
**(acl2->rcd x) = (if (badp x) (list (cons nil x)) x))**


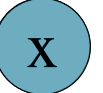

**(rcd->acl2 x) = (if (badp x) (cdar x) x))**


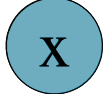

(acl2->rcd ) = 

(rcd->acl2 ) = 

(acl2->rcd ) = '((nil . ) = 

(rcd->acl2 ) = (rcd->acl2 '((nil . ) = 

(acl2->rcd ) = '((nil . ) = 

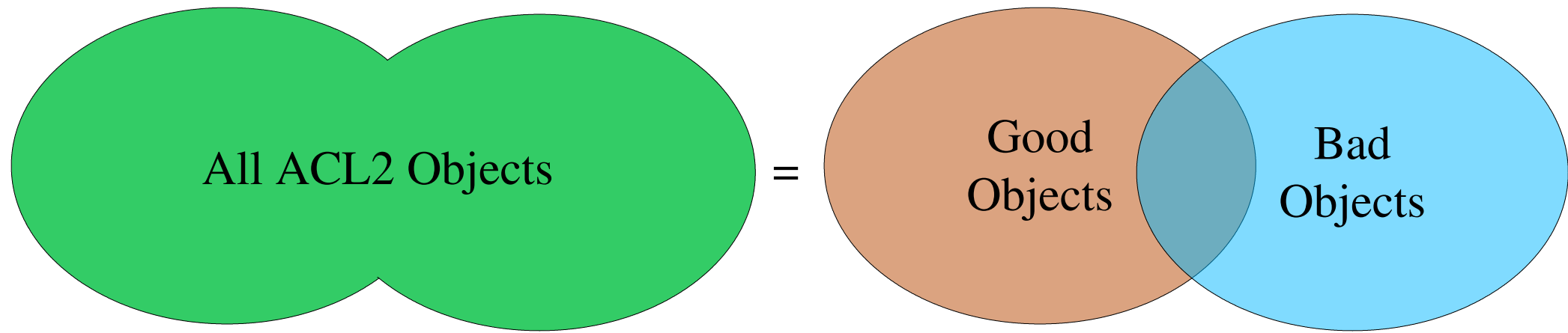
(rcd->acl2 ) = (rcd->acl2 '((nil . ) = 

All ACL2 Objects

=

Good  
Objects

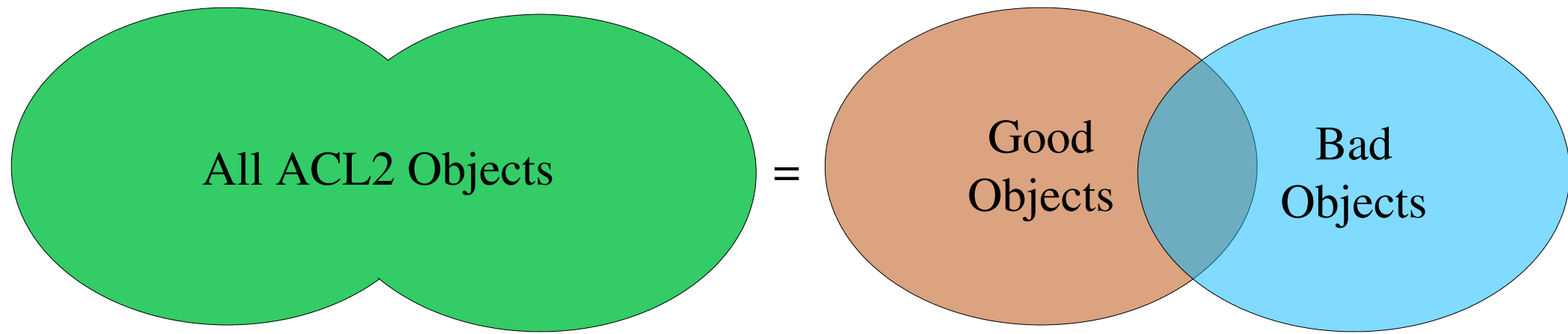
Bad  
Objects



# Resulting Theorems

- $(\text{recordp } (\text{acl2} \rightarrow \text{rcd } x))$
- $(\text{equal } (\text{rcd} \rightarrow \text{acl2 } (\text{acl2} \rightarrow \text{rcd } x)) x)$
- $(\text{implies } (\text{recordp } x)$

$(\text{equal } (\text{acl2} \rightarrow \text{rcd } (\text{rcd} \rightarrow \text{acl2 } x)) x)$



# The G and S Functions

- $(g\ a\ x) = (g\text{-aux}\ a\ (acl2\text{-}\>rcd\ x))$
- $(s\ a\ v\ x) = (rcd\text{-}\>acl2\ (s\text{-aux}\ a\ v\ (acl2\text{-}\>rcd\ x)))$
- If  $x$  is a good recordp, then these conversions are the identity function; we just call  $g\text{-aux}$  or  $s\text{-aux}$ .
- Otherwise, we find the “default record” for  $x$  and operate on it, and return the “uncoerced” result.



# Example 1

(s a (g a r) r)

==>

(rcd->acl2 (s-aux a

(g-aux a (acl2->rcd r))  
(acl2->rcd r)))

==>

(rcd->acl2 (acl2->rcd r))

==>

r

# Example 2

(s a x (s a y r))

==>

(rcd->acl2 (s-aux a x  


(accl2->rcd (rcd->acl2 (s-aux a y (acl2->rcd r)))))))))

==>

(rcd->acl2 (s-aux a x (s-aux a y 
 (acl2->rcd r))))))

==>

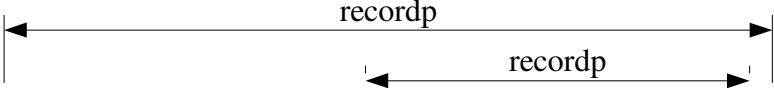
(rcd->acl2 (s-aux a x (acl2->rcd r)))

which is exactly (s a x r)

# Example 3

(g a (s a v r))

==>

(g-aux a (acl2->rcd  
  
(rcd->acl2 (s-aux a v (acl2->rcd r))))))

==>

(g-aux a (s-aux a v (acl2->rcd r))  


==>

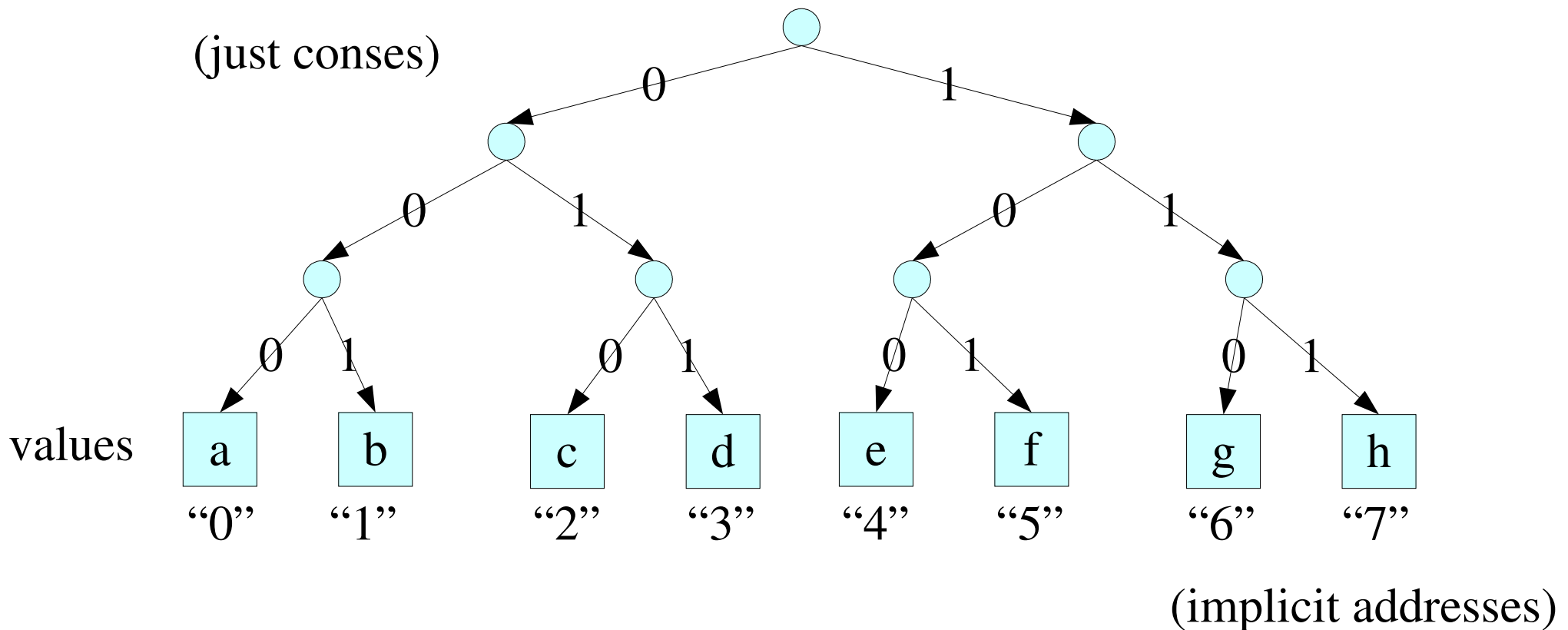
v

# Execution Efficiency Notes

- We pay a premium for these theorems when it comes to execution efficiency:
  - recordp, g-aux, s-aux are  $O(n)$  in the number of keys
  - badp calls recordp, so it is  $O(n)+$
  - acl2->rcd and rcd->acl2 call badp, so they are  $O(n)+$
  - g calls acl2->rcd, g-aux, so it is  $2*O(n)+$
  - s calls acl2->rcd, rcd->acl2, s-aux, so it is  $3*O(n)+$

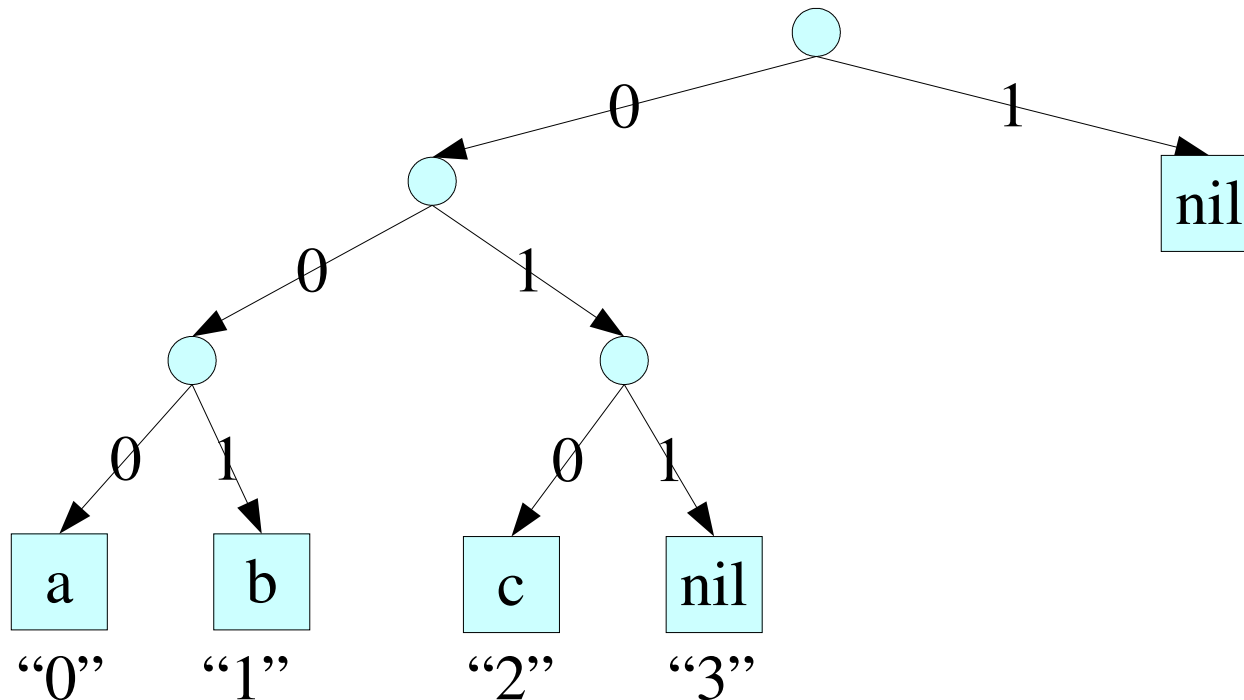
# Memtrees

- Bind  $2^n$  values to the “names”  $0, 1, 2, \dots, 2^n - 1$
- $O(\log_2 n)$  access, updates



# Canonical Memtrees

- Nil is our default value, collapse all-nil trees into a single nil
  - Canonical form for “equal” reasoning
  - Large space savings if values are sparse



# Memtree Operations (1)

```
(defun _memtree-p (mtree depth)
  (declare (xargs :guard (natp depth)))
  (if (zp depth)
      t
      (if (atom mtree)
          (null mtree)
          (and (_memtree-p (car mtree) (1- depth))
                (_memtree-p (cdr mtree) (1- depth))
                (not (and (null (car mtree))
                          (null (cdr mtree))))))))))
```

```
(defun _memtree-load (addr mtree depth)
  (declare (xargs :guard ...))
  (if (zp depth)
      mtree
      (_memtree-load (floor addr 2)
                     (if (= (mod addr 2) 0)
                         (car mtree)
                         (cdr mtree))
                     (1- depth))))
```

# Memtree Operations (2)

```
(defun _memtree-store (addr elem mtree depth)
  (declare (xargs :guard ...))
  (if (zp depth)
      elem
      (let ((quotient (floor addr 2)))
        (if (= (mod addr 2) 0)
            (cons (_memtree-store quotient elem (car mtree)
                                   (1- depth))
                  (cdr mtree))
            (cons (car mtree)
                  (_memtree-store quotient elem (cdr mtree)
                                   (1- depth))))))))
```



# Memtree Operations (3)

```
(defun _memtree-store-nil (addr mtree depth)
  (declare (xargs :guard ...))
  (if (zp depth)
      nil
      (if (atom mtree)
          nil
          (let ((quotient (floor addr 2)))
              (if (= (mod addr 2) 0)
                  (let ((left (_memtree-store-nil quotient
                                                       (car mtree)
                                                       (1- depth)))
                        (right (cdr mtree)))
                      (if (and (null left) (null right))
                          nil ;; collapse to canonical form
                          (cons left right)))
                  (let ((left (car mtree))
                        (right (_memtree-store-nil quotient
                                                       (cdr mtree)
                                                       (1- depth))))
                      (if (and (null left) (null right))
                          nil ;; collapse to canonical form
                          (cons left right)))))))
```

# Execution Efficiency Notes

- Optimizations with MBE and Guards
  - Multiple versions of each function, each logically equivalent, but...
  - When depth becomes small enough, we use a purely fixnum version of the function
  - Logand and ash are used rather than mod and floor in the executable counterparts (compiled into C's “&” and “>>” operations by GCL; very fast!)
  - Floor and mod used in :logic to take advantage of arithmetic-3/floor-mod reasoning

# Reasoning about Memtrees

- Memtrees must be well formed, like recordp
- Addresses (i.e., keys) are now also limited!
- Ugly depth parameter occurs throughout our theorems
- With fixing, we can get somewhat close to the record theorems, but not all the way.

# Reasoning about Memtrees (2)

```
(defthm _mmtree-load-same-store-1
  (implies
    (and (equal (_address-fix a depth)
                (_address-fix b depth))
         elem)
    (equal (_mmtree-load
            a
            (_mmtree-store b elem mtree depth)
            depth)
           elem)))
```

```
(defthm _mmtree-load-same-store-2
  (implies (equal (_address-fix a depth)
                  (_address-fix b depth))
    (equal (_mmtree-load
            a
            (_mmtree-store-nil b mtree depth)
            depth)
           nil)))
```

# Reasoning about Memtrees (3)

```
(defthm _mmtree-load-diff-store-1
  (implies (and (not (equal (_address-fix a depth)
                            (_address-fix b depth)))
              elem)
           (equal (_mmtree-load
                  a
                  (_mmtree-store b elem mtree depth)
                  depth)
                  (_mmtree-load a mtree depth))))
```

```
(defthm _mmtree-load-diff-store-2
  (implies (not (equal (_address-fix a depth)
                      (_address-fix b depth)))
           (equal (_mmtree-load
                  a
                  (_mmtree-store-nil b mtree depth)
                  depth)
                  (_mmtree-load a mtree depth))))
```

# Reasoning about Memtrees (4)

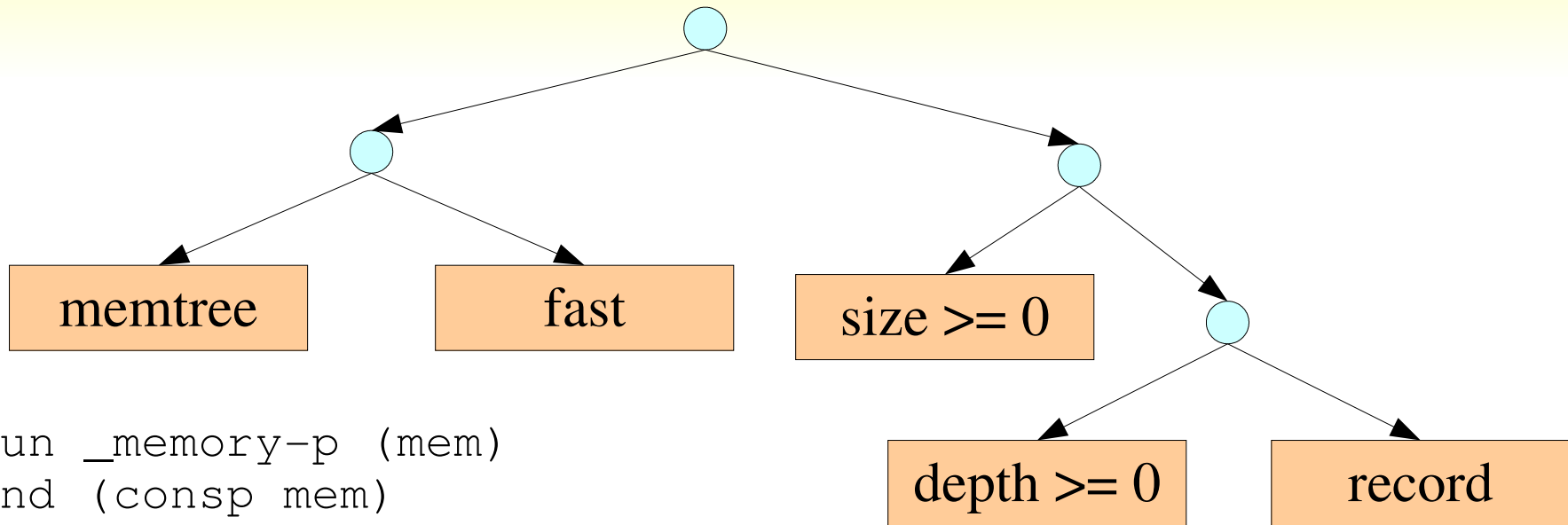
```
(defthm _memtree-store-same-load
  (implies (and (equal (_address-fix a depth)
                      (_address-fix b depth))
              (_memtree-load a mtree depth))
    (equal (_memtree-store
            a
            (_memtree-load b mtree depth)
            mtree
            depth)
           (_memtree-fix mtree depth))))
```

```
(defthm _memtree-store-same-load-nil
  (implies (and (equal (_address-fix a depth)
                      (_address-fix b depth))
              (not (_memtree-load a mtree depth)))
    (equal (_memtree-store-nil a mtree depth)
           (_memtree-fix mtree depth))))
```

# Memories

- The basic idea: glue a memtree to a record
  - Valid numeric addresses stored in the memtree part, other addresses stored in the record part, to fix address limitations
  - Depth of the tree can become part of the memory structure itself, so we won't need depth parameters all around
  - Can we use another invertible mapping to get the “real” record theorems?
  - Can we use guards to get good execution efficiency?

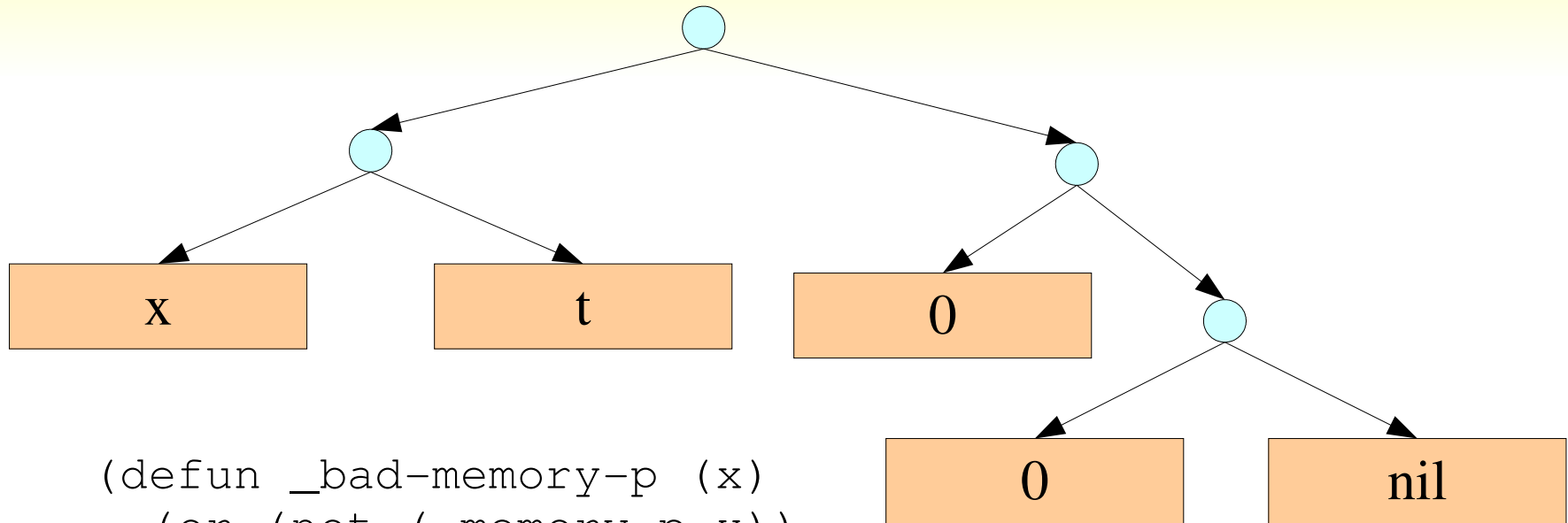
# Memories



```
(defun _memory-p (mem)
  (and (consp mem)
        (consp (car mem))
        (consp (cdr mem))
        (consp (cddr mem))
        (let ((mtree (caar mem))
              (fast (cdar mem))
              (size (cadr mem))
              (depth (caddr mem)))
          (and (natp size)
                (natp depth)
                (booleanp fast)
                (implies fast (signed-byte-p 29 depth))
                (<= size (expt 2 depth))
                (_memtree-p mtree depth))))))
```



# Bad Memories



```
(defun _bad-memory-p (x)
  (or (not (_memory-p x))
      (and (equal (_memory-fast x) t)
           (equal (_memory-depth x) 0)
           (equal (_memory-size x) 0)
           (equal (_memory-record x) nil)
           (_bad-memory-p (_memory-mtree x))))))
```

All ACL2 Objects

=

Memories

Bad  
Memories

$(\_to\text{-mem } x) = (if (\_bad x) (list (cons x t) 0 0 nil) x)$

$(\_from\text{-mem } x) = (if (\_bad x) (caar x) x)$

$(\_to\text{-mem } \textcircled{x}) = \textcircled{x}$

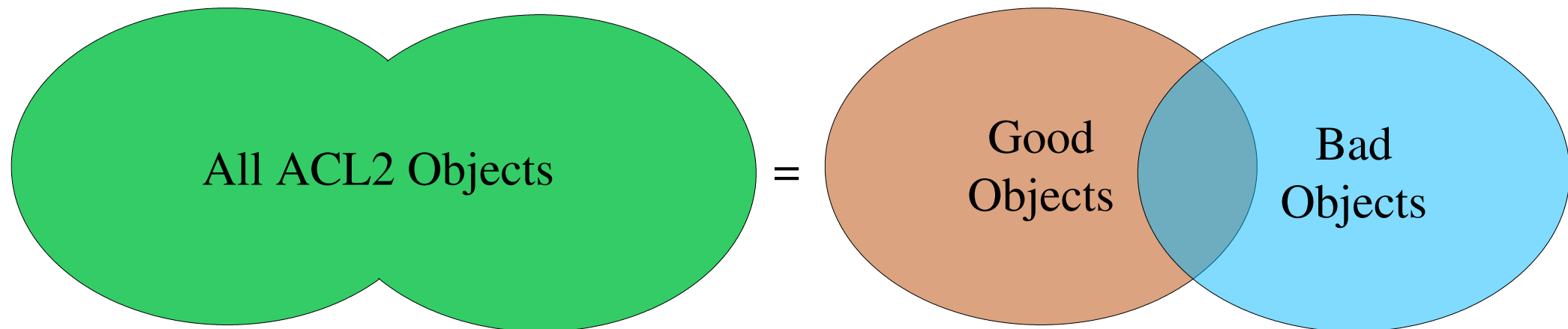
$(\_from\text{-mem } \textcircled{x}) = \textcircled{x}$

$(\_to\text{-mem } \textcircled{x}) = '(\dots \textcircled{x} \dots) = \textcircled{y}$

$(\_from\text{-mem } \textcircled{y}) = (\_from\text{-mem } '(\dots \textcircled{x} \dots)) = \textcircled{x}$

$(\_to\text{-mem } \textcircled{x}) = '(\dots \textcircled{x} \dots) = \textcircled{y}$

$(\_from\text{-mem } \textcircled{y}) = (\_from\text{-mem } '(\dots \textcircled{x} \dots)) = \textcircled{x}$



# Raw Memory Operations

```
(defun new (size)
  (declare (xargs :guard (posp size)))
  (if (or (not (posp size))
        (equal size 1))
      (cons (cons nil t) (cons 1 (cons 1 nil)))
      (let ((depth (_log2 (1- size))))
        (cons
         (cons nil (signed-byte-p 29 depth))
         (cons size
               (cons depth nil)))))))
```

```
(defun _load (addr mem)
  (declare (xargs :guard ...))
  (let ((mtree (_memory-mtree mem))
        (depth (_memory-depth mem))
        (record (_memory-record mem)))
    (if (address-p addr mem)
        (_memtree-load addr mtree depth)
        (g addr record))))
```

# Raw Memory Operations (2)

```
(defun _store (addr elem mem)
  (declare (xargs :guard ...))
  (let ((fast      (_memory-fast mem))
        (mtree    (_memory-mtree mem))
        (size      (_memory-size mem))
        (depth     (_memory-depth mem))
        (record    (_memory-record mem)))
    (if (address-p addr mem)
        (cons (cons (if elem
                     (_mmtree-store addr elem mtree depth)
                     (_mmtree-store-nil addr mtree depth))
                 fast)
              (cons size (cons depth record)))
        (cons (cons mtree fast)
              (cons size (cons depth (s addr elem record)))))))
```

# Final Memory Operations

```
(defun load (addr mem)
  (_load addr (_to-mem mem)))
```

```
(defun store (addr elem mem)
  (_from-mem (_store addr elem (_to-mem mem))))
```

```
(equal (load a (store a elem mem))
  elem))
```

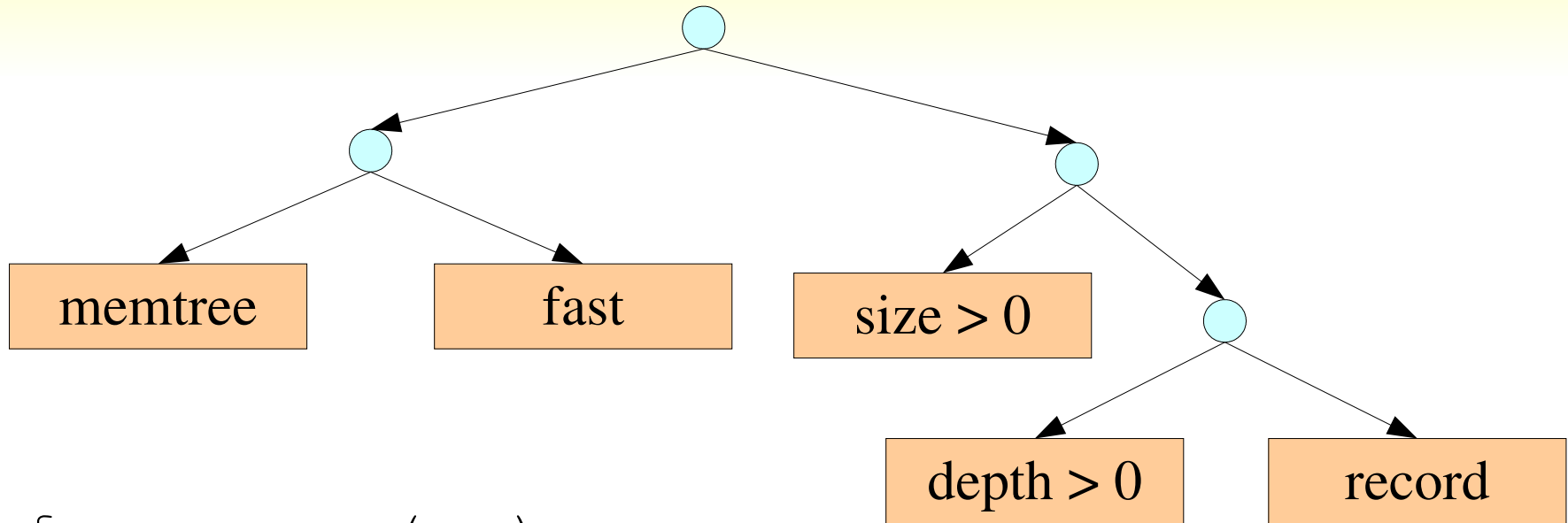
```
(implies (not (equal a b))
  (equal (load a (store b elem mem))
  (load a mem))))
```

```
(equal (store a e1 (store a e2 mem))
  (store a e1 mem)))
```

```
(implies (not (equal a b))
  (equal (store a e1 (store b e2 mem))
  (store b e2 (store a e1 mem)))))
```

```
(equal (store a (load a mem) mem)
  mem))
```

# Good Memories



```
(defun memory-p (mem)  
  (and (_memory-p mem)  
        (posp (_memory-size mem))  
        (posp (_memory-depth mem))))
```

All ACL2 Objects

=

Good  
Memories

Memories

Bad  
Memories

# Optimizations

```
(defun _load (addr mem)
  (declare (xargs :guard (and (memory-p mem)
                               (address-p addr mem))))
  (mbe :logic (let ((mtree (_memory-mtree mem))
                    (depth (_memory-depth mem))
                    (record (_memory-record mem)))
              (if (address-p addr mem)
                  (_mmtree-load addr mtree depth)
                  (g addr record))))

  :exec (let* ((fast (cdar mem))
              (mtree (caar mem))
              (depth (caddr mem)))
          (if fast
              (_fixnum-mmtree-load addr mtree depth)
              (_mmtree-load addr mtree depth))))))
```

```

(defun _store (addr elem mem)
  (declare (xargs :guard (and (memory-p mem)
                               (address-p addr mem))))
  (mbe :logic (let ((fast (_memory-fast mem))
                    (mtree (_memory-mtree mem)))
                (if (address-p addr mem)
                    (cons (cons
                          (if elem
                              (_mmtree-store addr elem mtree depth)
                              (_mmtree-store-nil addr mtree depth)) fast)
                          (cons size (cons depth record)))
                          ...))

                :exec (let* ((mtree (caar mem))
                             (fast (cdar mem))
                             (memcdr (cdr mem)) ...))

                (cons (cons (if fast
                              (if elem
                                  (_fixnum-mmtree-store addr elem mtree depth)
                                  (_fixnum-mmtree-store-nil addr mtree depth))
                              (if elem
                                  (_mmtree-store addr elem mtree depth)
                                  (_mmtree-store-nil addr mtree depth)))
                              fast)
                      memcdr))))

```





# Store

```
(defun store (addr elem mem)
  (declare (xargs :guard (and (memory-p mem)
                               (address-p addr mem))))
  (mbe :logic (_from-mem (_store addr elem (_to-mem mem)))
       :exec
       (let* ((mtree (caar mem))
              (fast  (cdar mem))
              (memcdr (cdr mem))
              (depth  (cadr memcdr)))
         (cons (cons (if fast
                       (if elem
                           (_fixnum-memtree-store addr elem mtree depth)
                           (_fixnum-memtree-store-nil addr mtree depth))
                       (if elem
                           (_memtree-store addr elem mtree depth)
                           (_memtree-store-nil addr mtree depth)))
                fast)
               memcdr))))))
```

# Guard Verification Theorems

```
(defthm load-new
  (equal (load addr (new size))
         nil))
```

```
(defthm store-memory
  (implies (memory-p mem)
           (memory-p (store addr elem mem))))
```

```
(defthm store-size
  (implies (memory-p mem)
           (equal (size (store addr elem mem))
                  (size mem))))
```

# Performance Results

Dimebox; GCL 2.6.6; ACL2 2.9.2

## 8-bit Memories

Average 10,580,912 loads/second

Average 1,928,895 stores/second

## 16-bit Memories

Average 4,694,484 loads/second

Average 598,711 stores/second

## 32-bit Memories

Average 1,176,470 loads/second

Average 293,255 stores/second

## 64-bit Memories

Average 294,986 loads/second

Average 108,459 stores/second

## 64-bit Memories (on Allegro)

Average 437,636 loads/second

Average 101,522 stores/second

# Final Comments

- Not convinced that equal is a good idea.
- Lots of thought required for properties that are easy to do with equivalences.
- Mandatory nature of canonical form limits options for other extensions.
- Records have really weird properties:
  - S affects the domain in weird ways, e.g., removes a key if value is nil, inserts a key otherwise
  - Weird behavior:  $(s 'a nil 3) = 3$ , so using S to erase keys doesn't necessarily get you a smaller record
  - This makes it harder to recur over records