Scalable Normalization of Heap Manipulating Functions

David Greve
November 2007

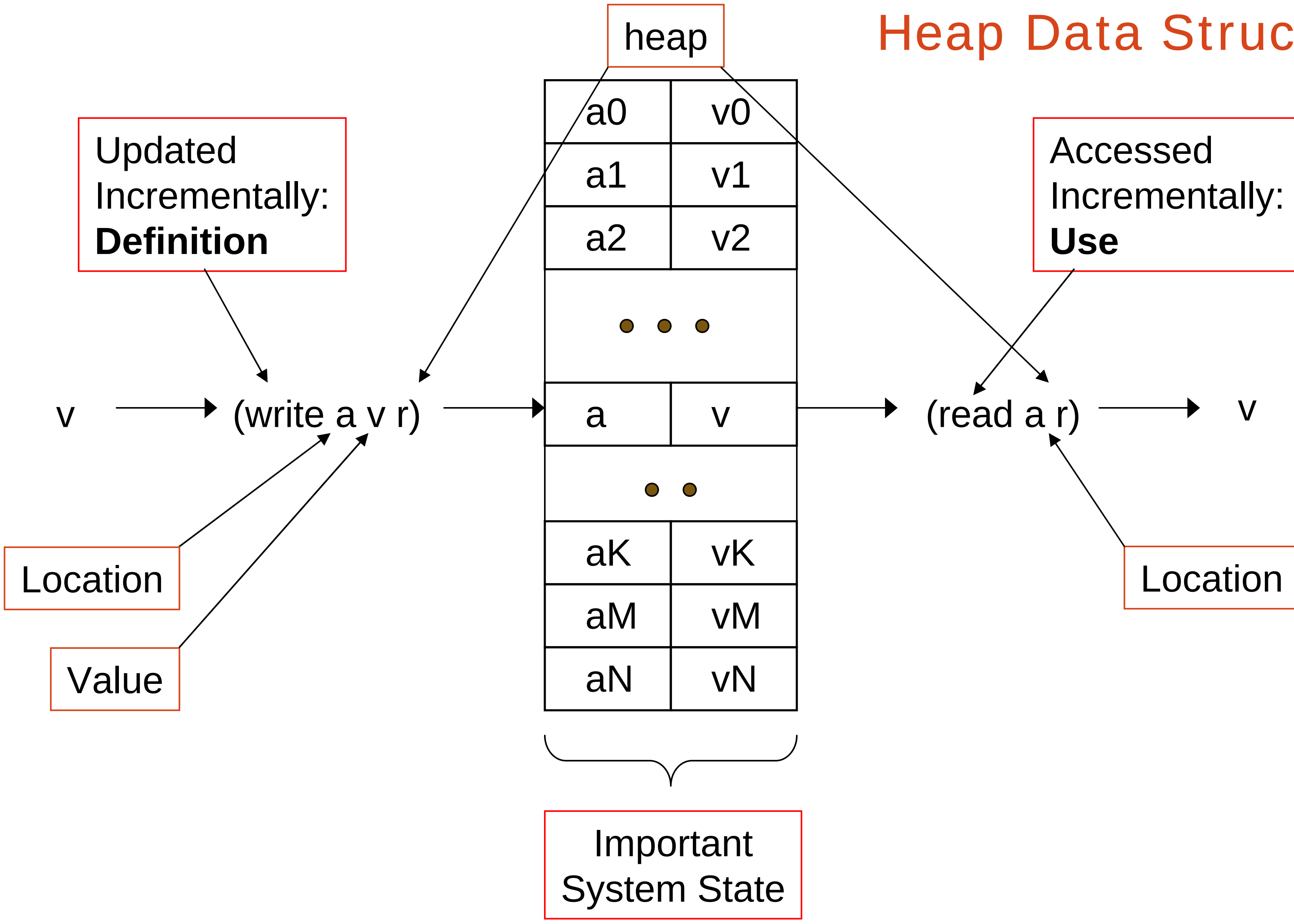
Rockwell Collins

Heap Data Structures

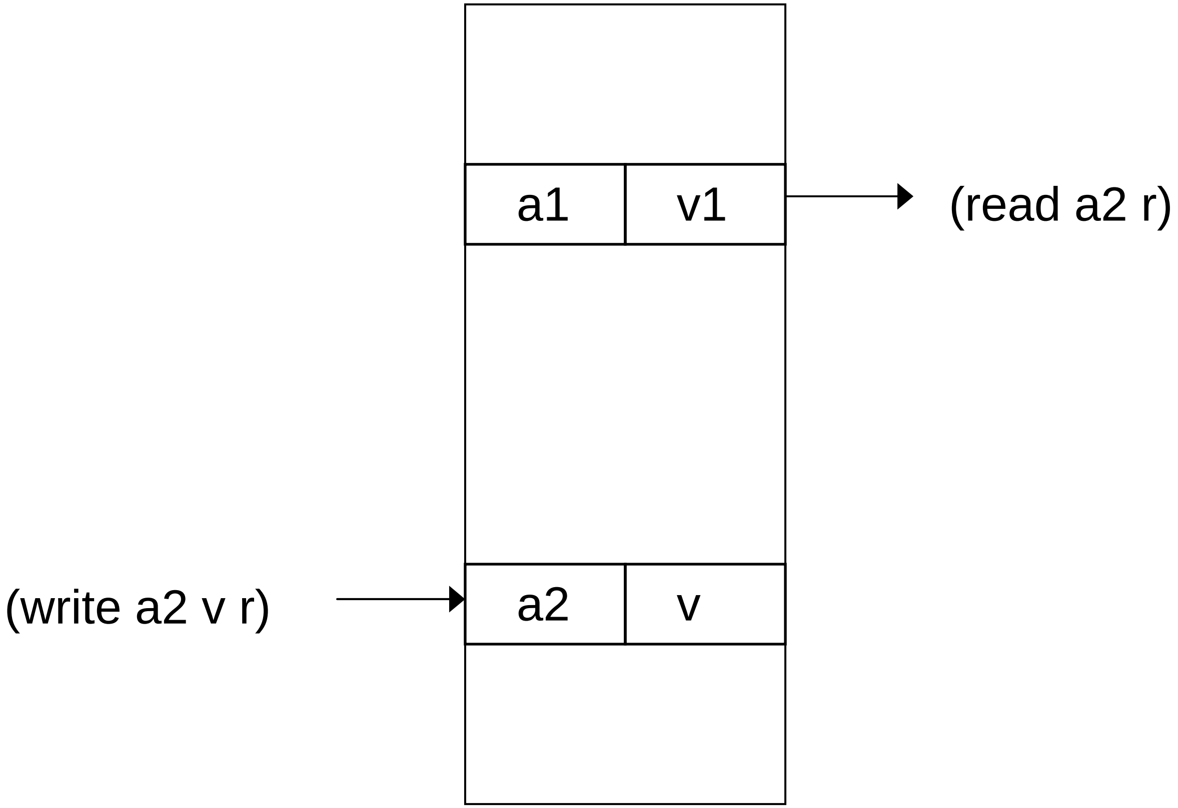
heap

a0	v0
a1	v1
a2	v2
• • •	
a	v
• •	
aK	vK
aM	vM
aN	vN

Heap Data Structures



Non-Interference




```
(defthm read-over-write-non-interference
  (implies
    (not (equal a1 a2))
    (equal (read a1 (write a2 v x))
                (read a1 x))))
```

Read/Write Towers

```
(defun write_3 (a v r)
  (write_2 a v r))

(defun write_2 (a v r)
  (write_1 a v r))

(defun write_1 (a v r)
  (write a v r))

(write a v r)
```

```
(defun read_3 (a r)
  (read_2 a r))

(defun read_2 (a r)
  (read_1 a r))

(defun read_1 (a r)
  (read a r))

(read a r)
```

Read/Write Towers

```
(defun write_3 (a v r)
  (write_2 a v r))

(defun write_2 (a v r)
  (write_1 a v r))

(defun write_1 (a v r)
  (write a v r))

(write a v r)
```

```
(defun read_3 (a r)
  (read_2 a r))

(defun read_2 (a r)
  (read_1 a r))

(defun read_1 (a r)
  (read a r))

(read a r)
```

(write a v r) ←— read-over-write-non-interference —→ (read a r)

Read/Write Towers

```
(defun write_3 (a v r)
  (write_2 a v r))

(defun write_2 (a v r)
  (write_1 a v r))

(defun write_1 (a v r)
  (write a v r))

(write a v r)
```

```
(defun read_3 (a r)
  (read_2 a r))

(defun read_2 (a r)
  (read_1 a r))

(defun read_1 (a r)
  (read a r))

(read a r)
```

read-over-write-1-non-interference

Read/Write Towers

(defun write_3 (a v r)
  (write_2 a v r))

(defun read_3 (a r)
  (read_2 a r))

(defun write_2 (a v r)
  (write_1 a v r))

(defun read_2 (a r)
  (read_1 a r))

read-1-over-write-non-interference

(defun write_1 (a v r)
  (write a v r))

(defun read_1 (a r)
  (read a r))

(write a v r)

(read a r)

Read/Write Towers

```
(defun write_3 (a v r)
  (write_2 a v r))

(defun write_2 (a v r)
  (write_1 a v r))

(defun write_1 (a v r)
  (write a v r))

(write a v r)
```

```
(defun read_3 (a r)
  (read_2 a r))

(defun read_2 (a r)
  (read_1 a r))

(defun read_1 (a r)
  (read a r))

(read a r)
```

read-1-over-write-1-non-interference

2 x 2 = 4 rules

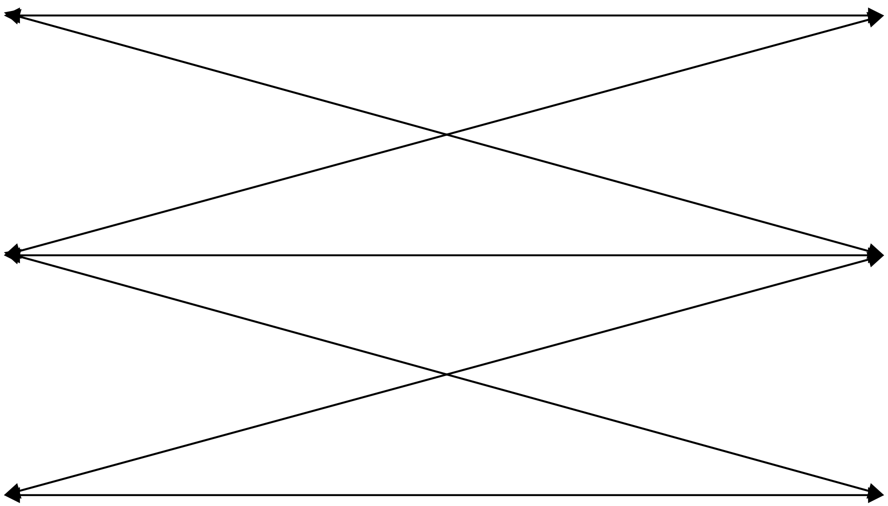
Read/Write Towers

```
(defun write_3 (a v r)
  (write_2 a v r))

(defun write_2 (a v r)
  (write_1 a v r))

(defun write_1 (a v r)
  (write a v r))

 (write a v r)
```



```
(defun read_3 (a r)
  (read_2 a r))

(defun read_2 (a r)
  (read_1 a r))

(defun read_1 (a r)
  (read a r))

  (read a r)
```

Read/Write Towers

(defun write_3 (a v r)
  (write_2 a v r))

(defun read_3 (a r)
  (read_2 a r))

(defun write_2 (a v r)
  (write_1 a v r))

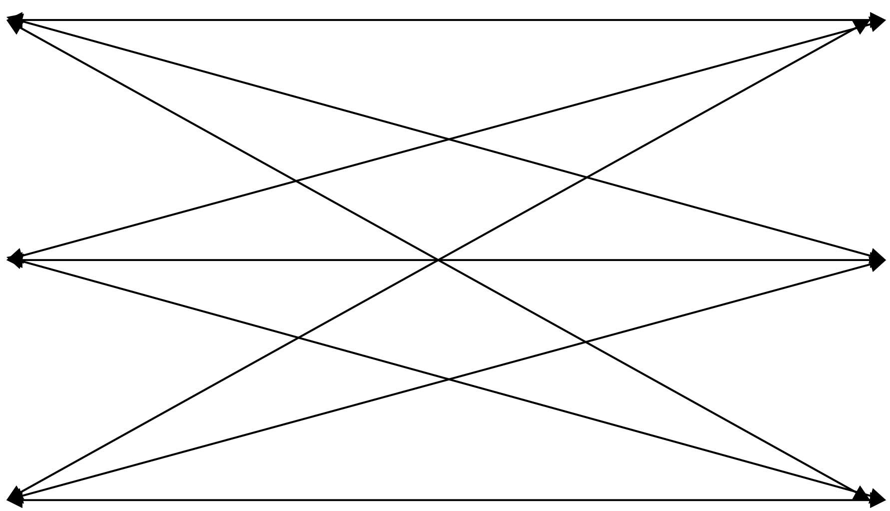
(defun read_2 (a r)
  (read_1 a r))

(defun write_1 (a v r)
  (write a v r))

(defun read_1 (a r)
  (read a r))

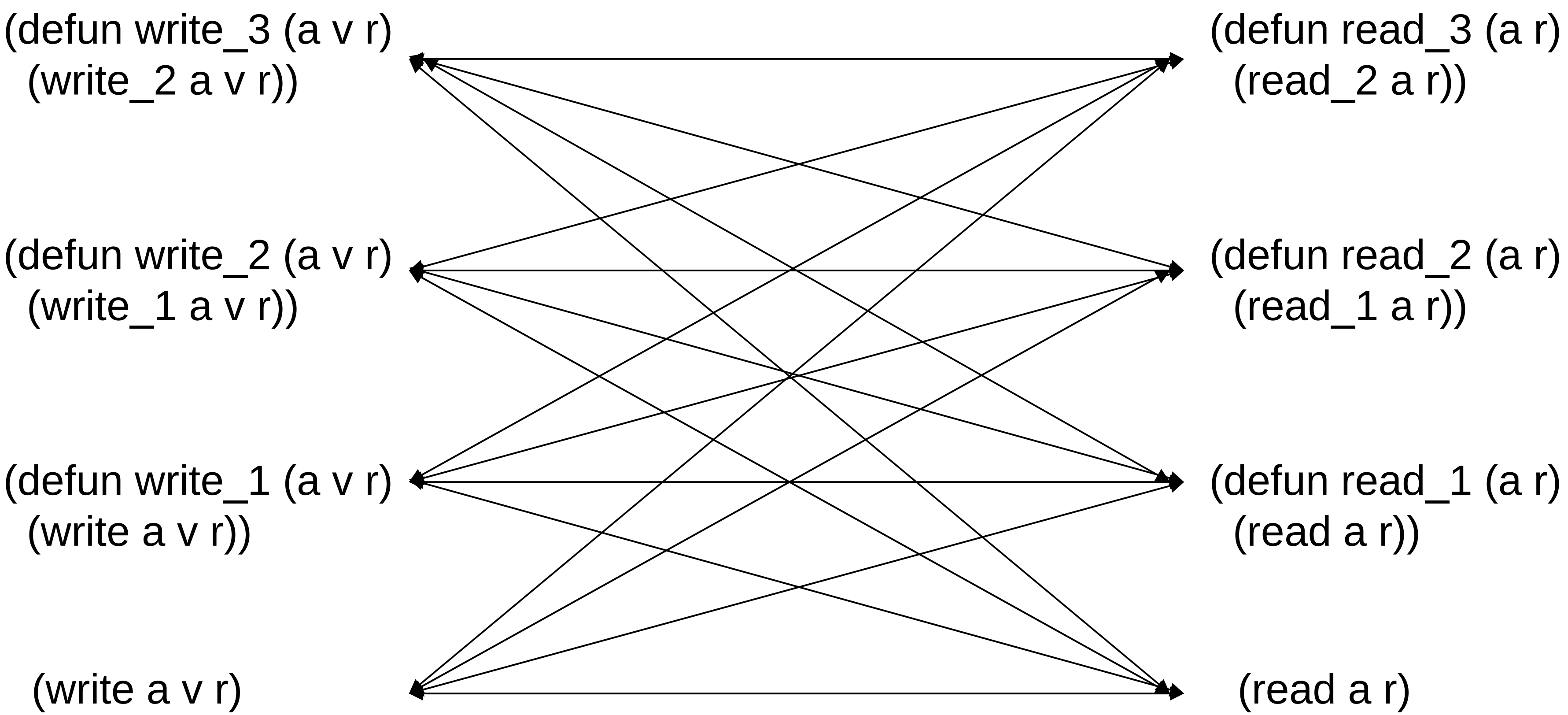
(write a v r)

(read a r)



3 x 3 = 9 rules

Read/Write Towers



N x N = 4 x 4 = 16 rules

Non-interference in complex systems

- Complex Systems
  - Hierarchical Design
  - Build larger components from many simpler components
- Compositional Verification Methodology Essential
  - Specify behavior once (locally)
  - Use behavior many times (globally)
- Non-interference
  - Not a complex property
  - Number of theorems is quadratic in total number of components
  - Standard Approach
    - Articulate property between every component
    - Not Compositional
    - Doesn't scale

And now for something completely different: Congruences

- Congruence-based Rewriting
  - Built-In to ACL2
  - Treats Certain Predicate Relations “just like equality”
  - Use Relations to Define Rewrite Rules

- Provides Strong Normalization
  - (Near) Minimal Representations

- Congruence-based Rewriting
  - More powerful than rewrite rules
  - More scalable than syntactic techniques (:meta / bind-free)

- Scalable
  - Defined Locally
  - Used Globally

Rewriting Context

- Obviously (cons x (cons x y)) is not equal to (cons x y),

```
(cons x (cons x y))
(cons x y)
```

- But they are equivalent in "the second argument of member"

```
(defthm member-cons-duplicates
  (iff (member a (cons x (cons x y)))
       (member a (cons x y))))
```

- So we can replace one with the other in that context

Defining a Rewriting Context

- ACL2 Generalizes this notion
  - "the second argument of member"

- Uses Equivalence Relations
  - Formalize essential properties of "the second argument of member"

```
(defun set-equiv (x y)
  (if (consp x)
      (and (member (car x) y)
           (set-equiv (cdr x) (remove (car x) y))
    (not (consp y))))
```

- Formally Introduced in ACL2 via defequiv
  - (defequiv set-equiv)
  - Associates equivalence relation with a rewriting context

Driver Rules

- Rewrite rules employing equivalence relations

```
(defthm set-equiv-cons-cons-driver
  (set-equiv (cons x (cons x y))
             (cons x y)))
```

  - Does not rewrite set-equiv to true
  - Replaces (cons x (cons x y)) with (cons x y)
  - In a **set-equiv** rewriting context

- Driver Rules
  - Concise, Automatic, Unconstrained
  - Enhanced Normalization

Congruences

- Driver Rules
  - Only Applied in specific rewriting contexts

- Congruence Rules
  - Establish rewriting contexts
  - Indicate when it is sound to use specified equivalence relations

```
(defthm set-equiv-implies-iff-in-2
  (implies
    (set-equiv x y)
    (iff (member a x) (member a y)))
  :rule-classes (:congruence))
```

Congruence-based Rewriting: Synopsys

- Rewriting contexts
  - Characterized by equivalence relations
- Driver Rules
  - Apply context-sensitive simplifications
- Congruence Rules
  - Chain from one context to another
- Congruence-based Rewriting
  - More powerful than rewrite rules
  - More scalable than syntactic techniques

```
(defequiv set-equiv)

(defthm set-equiv-cons-cons-driver
  (set-equiv (cons x (cons x y))
             (cons x y)))

(defcong set-equiv iff (member a x) 2)

(defcong set-equiv set-equiv (cons a x) 2)
```

↓

```
(defthm member-cons-duplicates
  (iff (member a (cons x (cons x y)))
       (member a (cons x y))))
```

Nary Congruences

- Nary Library
  - Extends ACL2 congruence capabilities
  - Enables parameterized equivalence relations and congruences
  - Used to define parameterized rewrite rules

```
(defun mod-equiv (x y n)
  (equal (mod x n)
         (mod y n)))
```

parameter

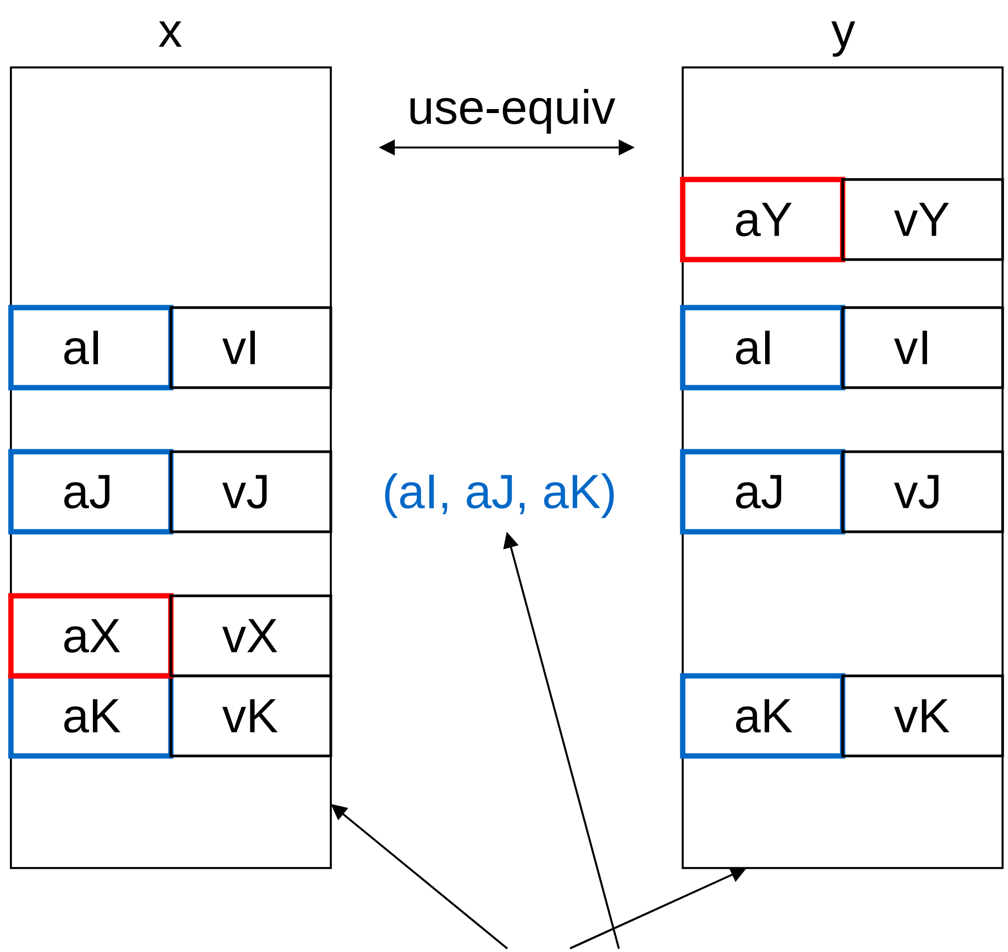
```
(defthm mod-reduction
  (mod-equiv (mod x n) x n))
```

Rewrites this .. | .. into this .. | .. in a "mod n" context.

Non-Interference as a Congruence

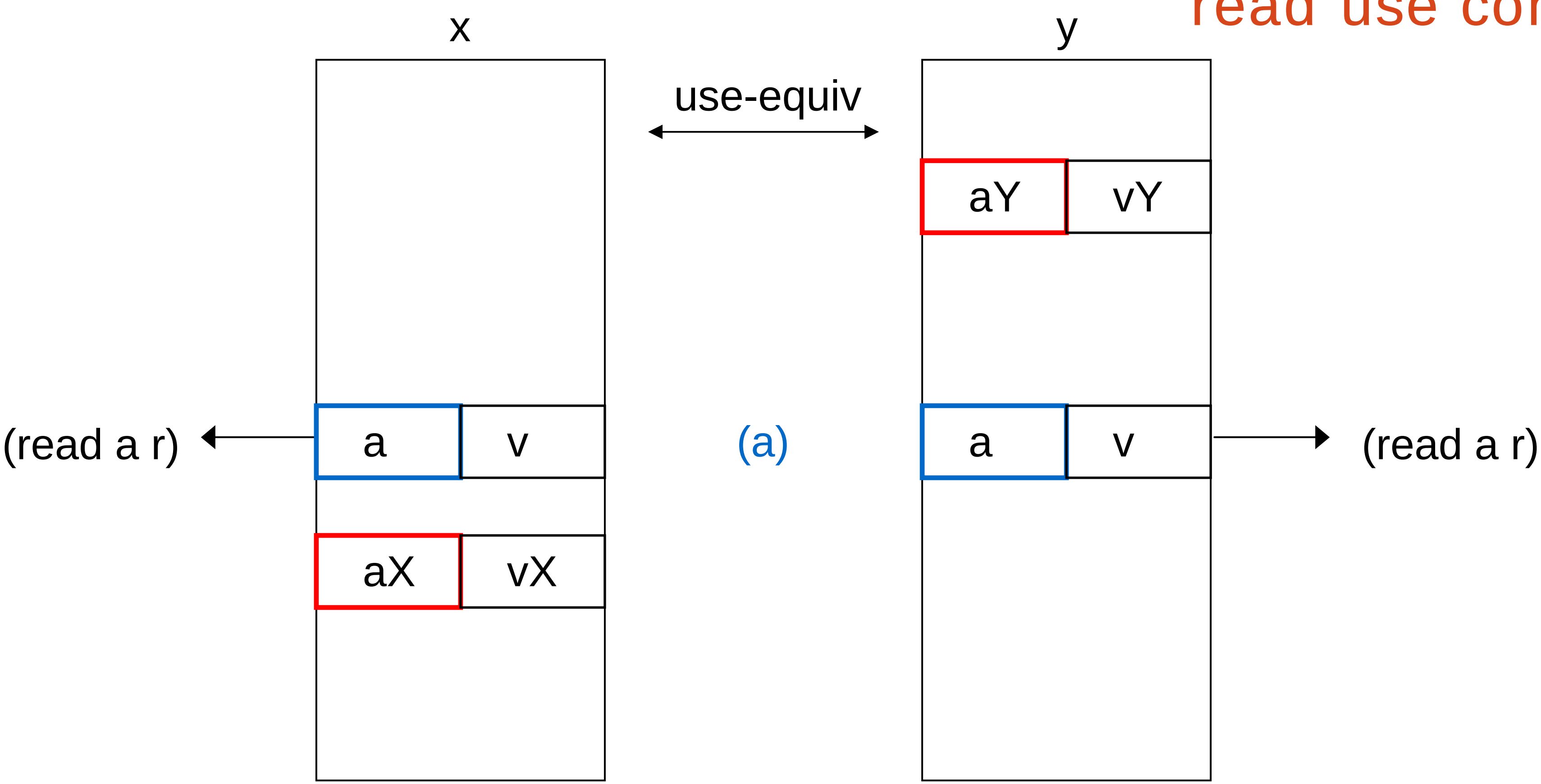
- Non-interference properties can be expressed via parameterized congruences
  - Given an appropriate equivalence relation
- Inherits Congruence Properties
  - Provides Strong Normalization
    - (Near) Minimal Representations
  - Scalable
    - Defined Locally
    - Used Globally

use-equiv




```
(defun use-equiv (x y list)
 (if (consp list)
  (and (equal (read (car list) x)
              (read (car list) y))
    (use-equiv x y (cdr list)))))
```

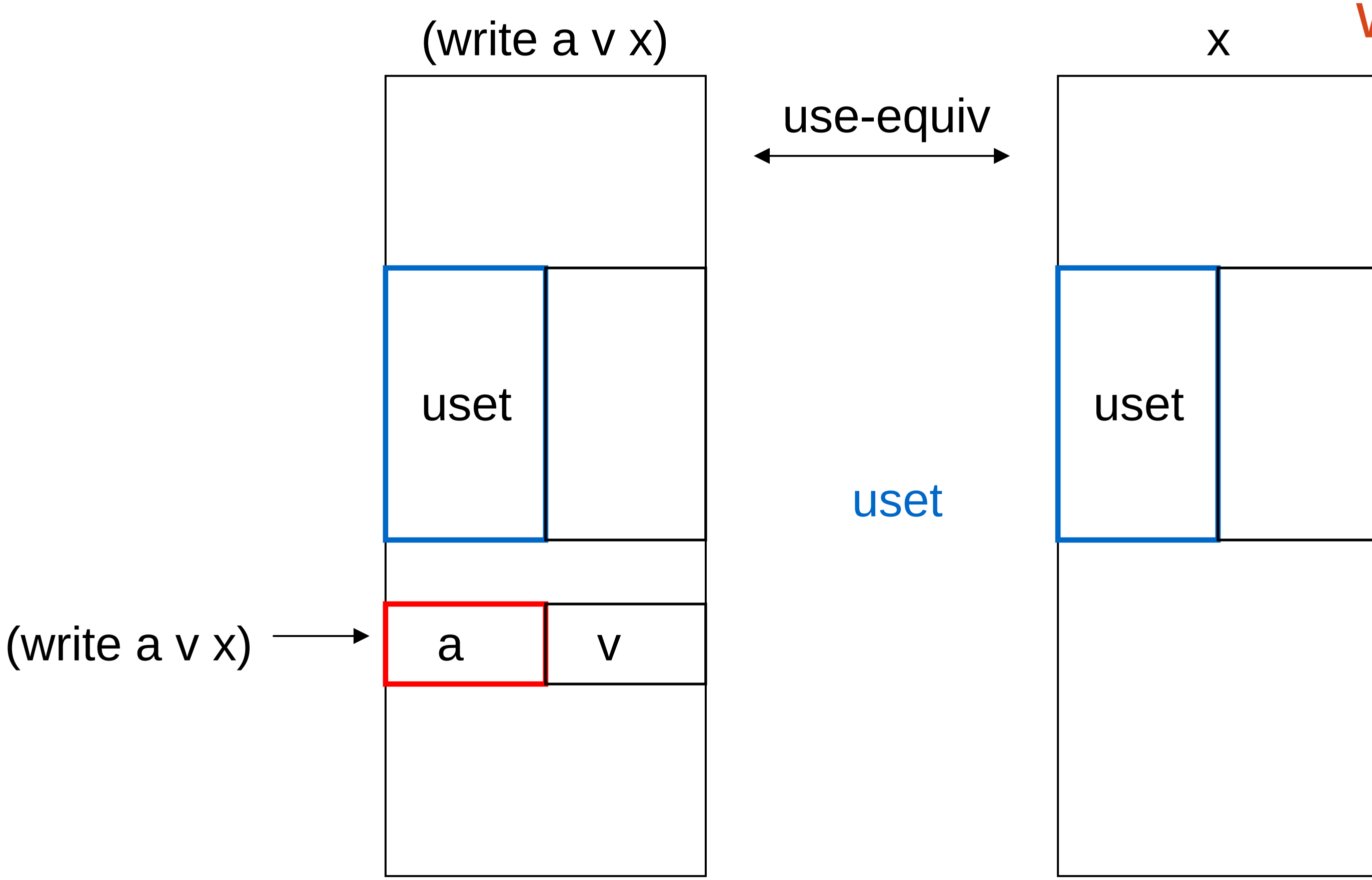
read use cong




```
(defthm read-use-cong
  (implies
    (use-equiv x y (list a))
    (equal (read a x)
                (read a y))))
:rule-classes (:nary-congruence))
```

Nary congruence rule

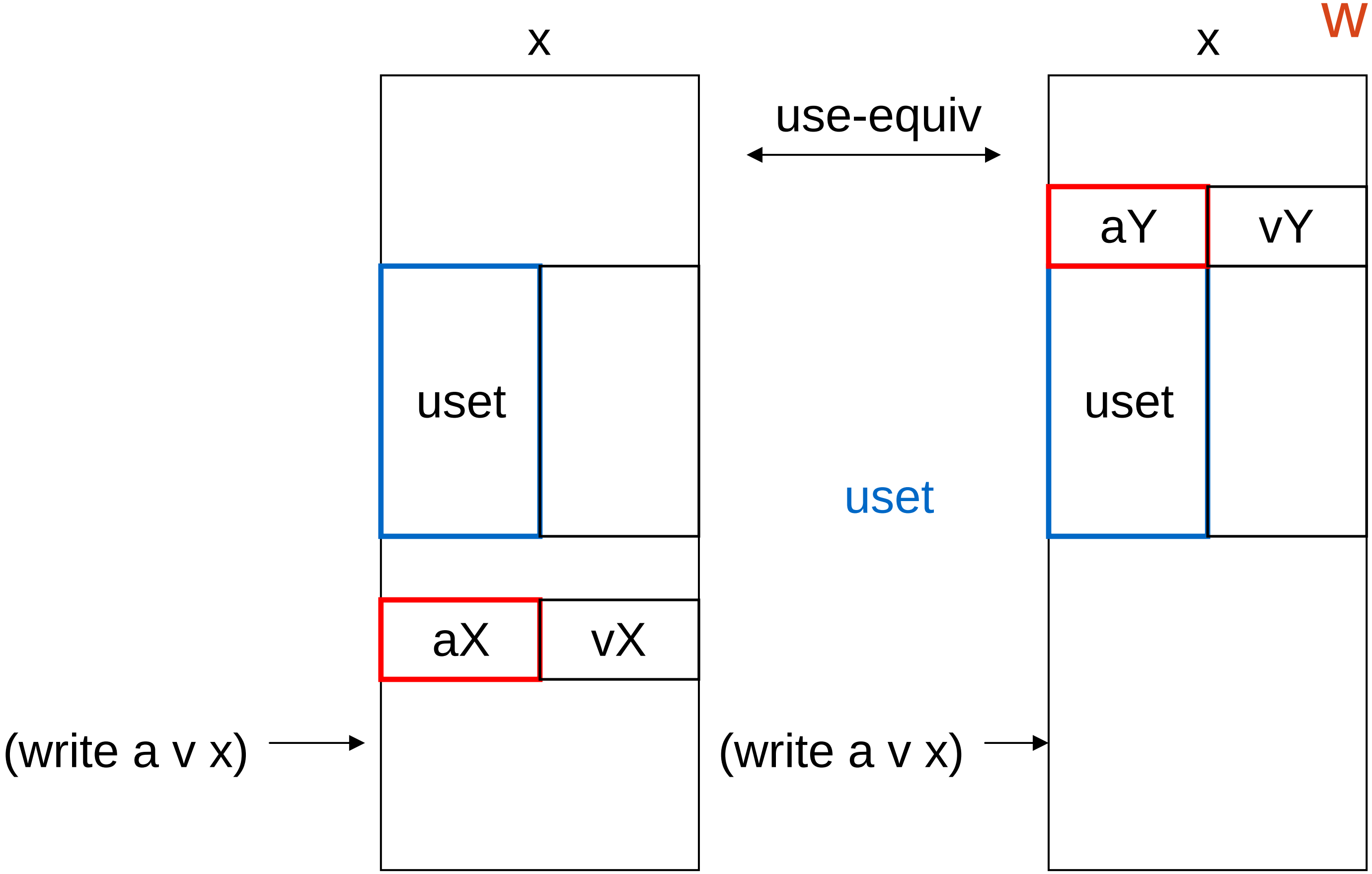
write use elim




```
(defthm write-use-elim
  (implies
    (not (member a uset))
    (use-equiv (write a v x)
               x
               uset)))
```

Nary driver rule

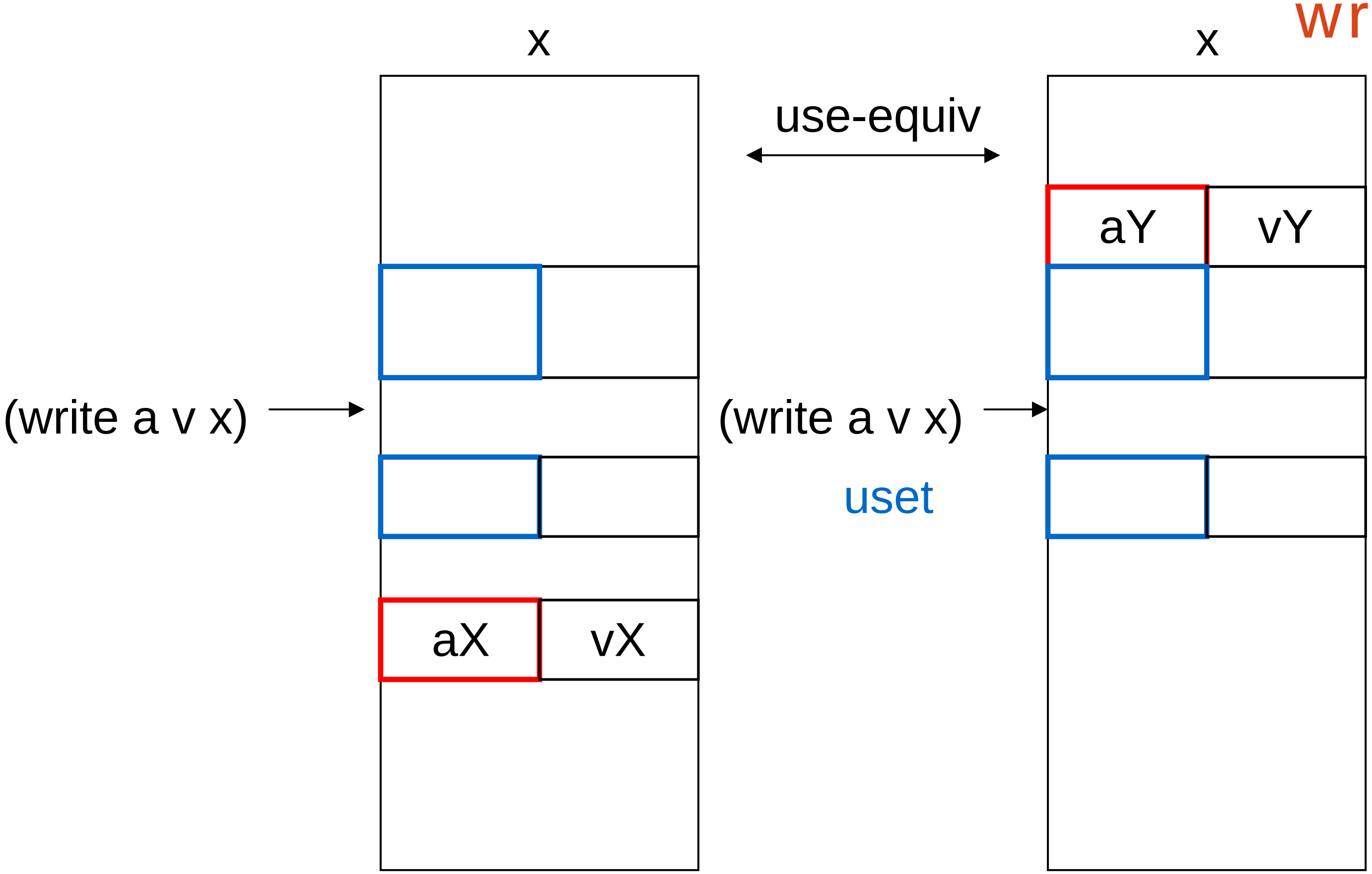
write use-cong




```
(defthm write-use-cong
  (implies
    (use-equiv x y uset)
    (use-equiv (write a v x)
               (write a v y)
               uset)))
```

Nary congruence rule

write use-cong!




```
(defthm write-use-cong
  (implies
   (use-equiv x y (remove a uset))
   (use-equiv (write a v x)
              (write a v y)
              uset)))
```

Nary congruence rule

Local Characterization

```
(defthm read-use-cong
  (implies
    (use-equiv x y (list a))
    (equal (read a x)
           (read a y))))

(defthm write-use-elim
  (implies
    (not (member a uset))
    (use-equiv (write a v x)
               x
               uset)))

(defthm write-use-cong
  (implies
    (use-equiv x y (remove a uset))
    (use-equiv (write a v x)
               (write a v y)
               uset)))
```

These three theorems characterize the non-interference properties of read and write operations via use-equiv

These three theorems are sufficient to characterize the non-interference properties of any function defined in terms of read and write.

Local characterization and global application: properties essential for scalable non-interference

Example Application

```
(defthm read-use-cong
  (implies
   (use-equiv x y (list a))
   (equal (read a x)
          (read a y))))
```

(list a)

```
(defthm read-over-write-normalization
  (implies
   (not (member a (list b c d)))
   (equal (read a (write b v1
                    (write c v2
                      (write a v3
                        (write d v4
                          (write a v5 x))))))
          (read a (write a v3 x)))))
```

Example Application

```
(list a)
```

```
(defthm read-over-write-normalization
  (implies
    (not (member a (list b c d)))
    (equal (read a (write b v1
                         (write c v2
                           (write a v3
                             (write d v4
                               (write a v5 x))))))
              (read a (write a v3 x)))))
```

```
(defthm write-use-elim
  (implies
    (not (member a uset))
    (use-equiv (write a v x)
               x
               uset)))
```

Example Application

(list a)

```
(defthm read-over-write-normalization
  (implies
    (not (member a (list b c d)))
    (equal (read a
                 (write c v2
                   (write a v3
                     (write d v4
                       (write a v5 x)))))
           (read a (write a v3 x)))))
```

```
(defthm write-use-elim
  (implies
    (not (member a uset))
    (use-equiv (write a v x)
               x
               uset)))
```

Example Application

(list a)

```
(defthm read-over-write-normalization
  (implies
    (not (member a (list b c d)))
    (equal (read a
                   (write a v3
                     (write d v4
                       (write a v5 x))))
           (read a (write a v3 x)))))
```

nil

```
(defthm write-use-cong
  (implies
    (use-equiv x y (remove a uset))
    (use-equiv (write a v x)
               (write a v y)
               uset)))
```

Example Application

```
(defthm read-over-write-normalization
  (implies
    (not (member a (list b c d)))
    (equal (read a
                   (write a v3
                     (write d v4
                       (write a v5 x))))
           (read a (write a v3 x)))))
```

```
(defthm write-use-elim
  (implies
    (not (member a uset))
    (use-equiv (write a v x)
               x
               uset)))
```

nil

Example Application

```
(defthm read-over-write-normalization
  (implies
    (not (member a (list b c d)))
    (equal (read a
                             (write a v3
nil
                                 (write a v5 x)))
          (read a (write a v3 x)))))
```

```
(defthm write-use-elim
  (implies
    (not (member a uset))
    (use-equiv (write a v x)
               x
               uset)))
```

Example Application

```
(defthm read-over-write-normalization
  (implies
    (not (member a (list b c d)))
    (equal (read a

nil                          (write a v3

                                        x))
           (read a (write a v3 x)))))
```

Tower Example

```
(defun write_3 (a v r)
  (write_2 a v r))

(defun write_2 (a v r)
  (write_1 a v r))

(defun write_1 (a v r)
  (write a v r))

(write a v r)
```

```
(defthm read_i-over-write_x-normalization
  (implies
    (not (member a (list b c d)))
    (equal (read_i a (write_j b v1
                              (write_k c v2
                                (write_x a v3
                                  (write_y d v4
                                    (write_z a v5 x))))))
           (read_i a (write_x a v3 x)))))
```

3 x N = 3 x 4 = 12 rules

```
(defun read_3 (a r)
  (read_2 a r))

(defun read_2 (a r)
  (read_1 a r))

(defun read_1 (a r)
  (read a r))

(read a r)
```

Read/Modify/Write

```
(defun move (rptr wptr r)
  (write wptr (read rptr r) r))

(defthm move-use-cong
  (implies
    (use-equiv x y (cons rptr (remove wptr uset)))
    (use-equiv (move rptr wptr x)
               (move rptr wptr y)
               uset)))

(defthm move-use-elim
  (implies
    (not (member wptr uset))
    (use-equiv (move rptr wptr x)
               x
               uset)))
```

Crawlers

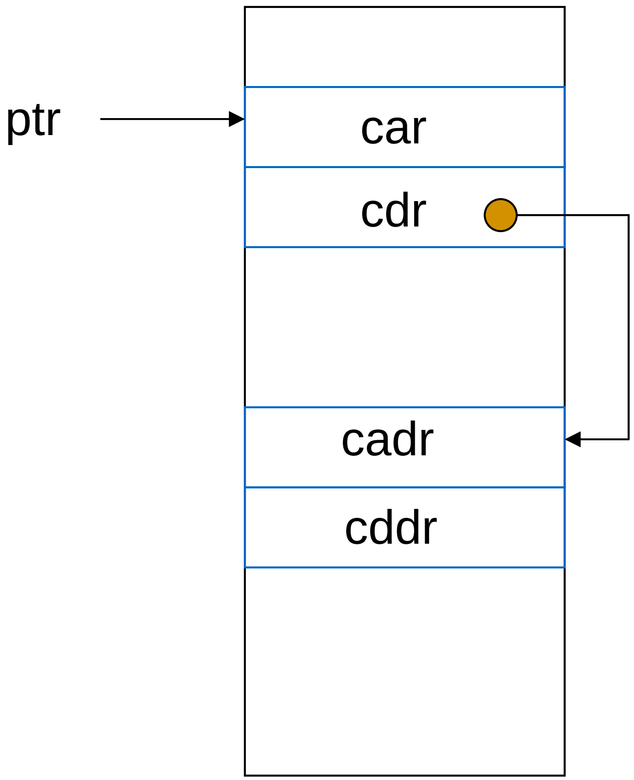
```
(defun get-cadr (ptr r)
  (read (read (+ ptr 1) r) r))

(defun get-cadr-uset (ptr r)
  (list (+ ptr 1) (read (+ ptr 1) r)))

(defthm get-cadr-use-cong
  (implies
    (use-equiv x y (get-cadr-uset ptr x))
    (equal (get-cadr ptr x)
           (get-cadr ptr y))))

(defthm get-cadr-uset-use-cong
  (implies
    (use-equiv x y (list (+ ptr 1)))
    (equal (get-cadr-uset ptr x)
           (get-cadr-uset ptr y))))
```

<u>Every</u> function of the heap can be characterized.



Parameterized Congruence for Non-Interference

- Non-interference properties can be expressed via parameterized congruences
  - use-equiv

- Inherits Congruence Properties
  - Provides Strong Normalization
    - (Near) Minimal Representations
  - Scalable
    - Defined Locally
    - Used Globally