Rewrite Rule Anatomy & Medicine

Recursion & Induction
Guest lecturer – Sol Swords (Arm Inc.)
March 11, 2025

Available at: http://bit.ly/3XSGPMj

Break down of a rewrite rule

```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

Break down of a rewrite rule

```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

Break down of a rewrite rule

Hypotheses

```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

Break down of a rewrite rule

Hypotheses

```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

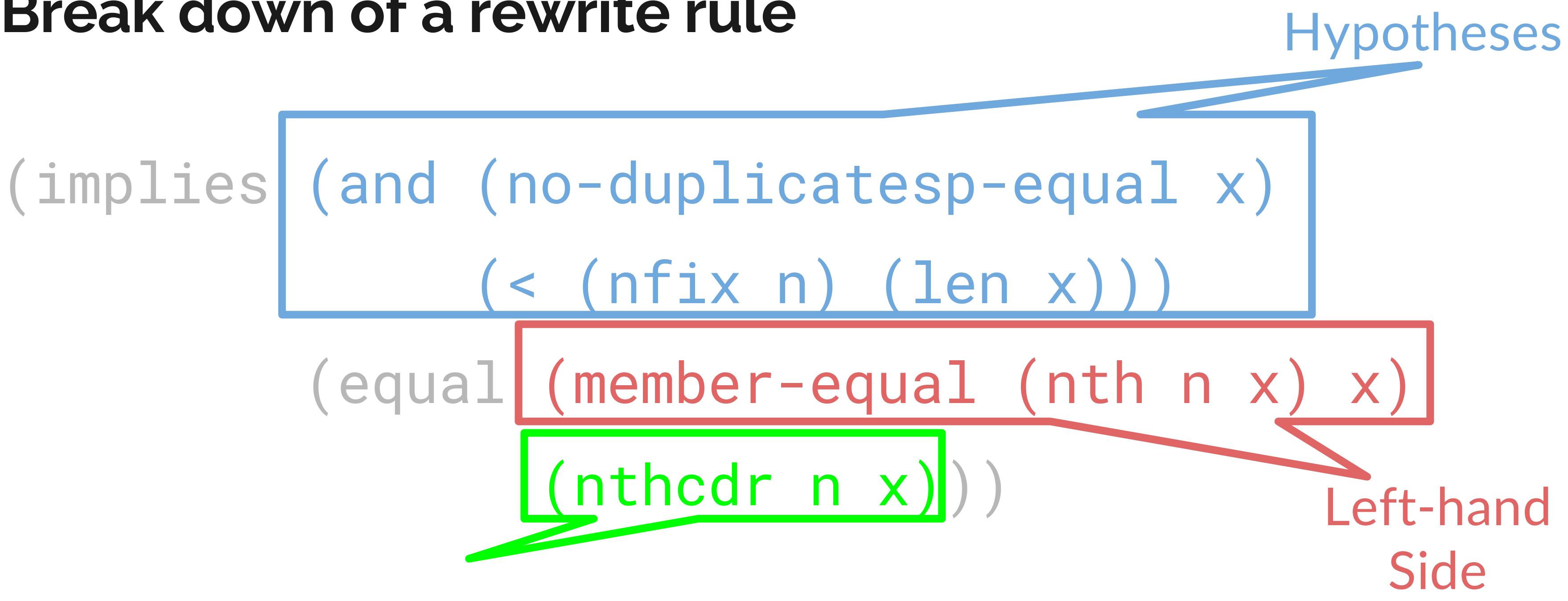
Break down of a rewrite rule

Hypotheses

```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

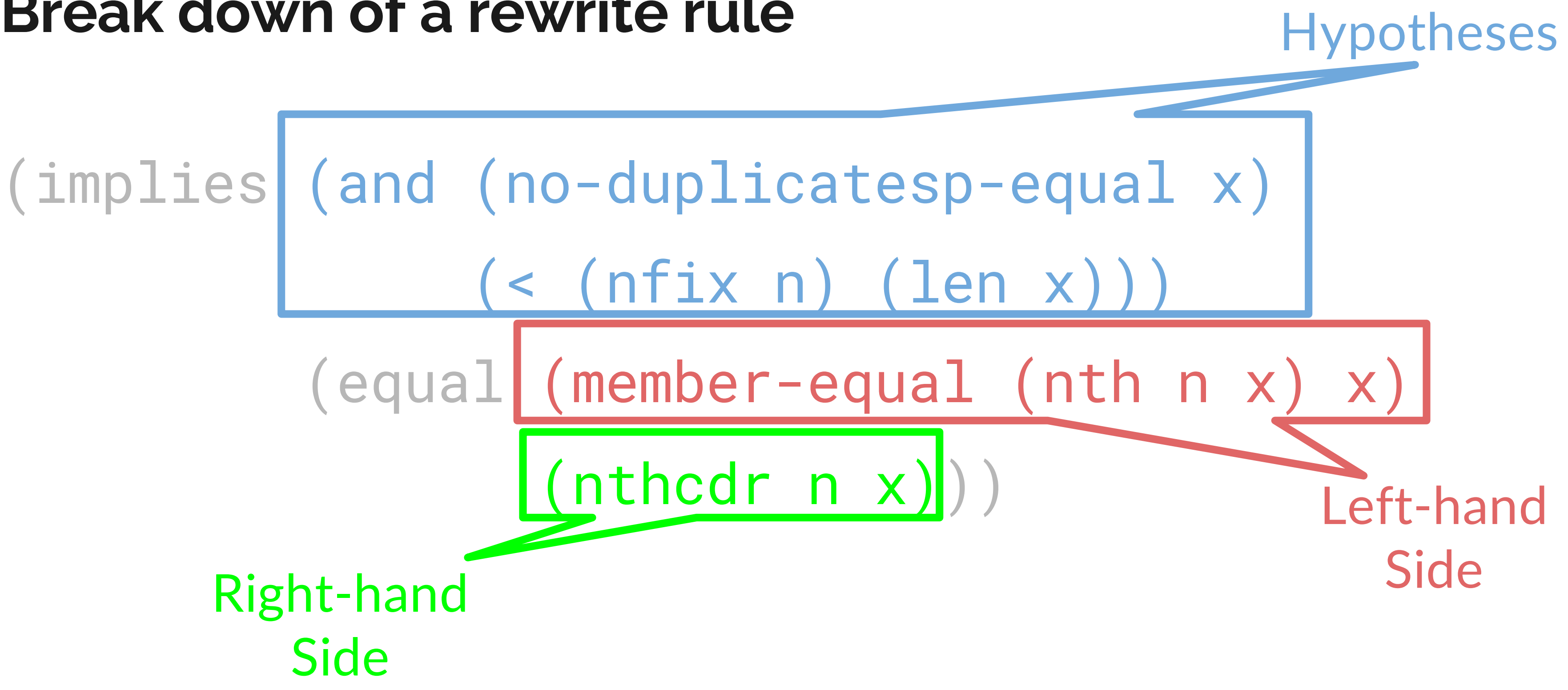
Left-hand Side

Break down of a rewrite rule



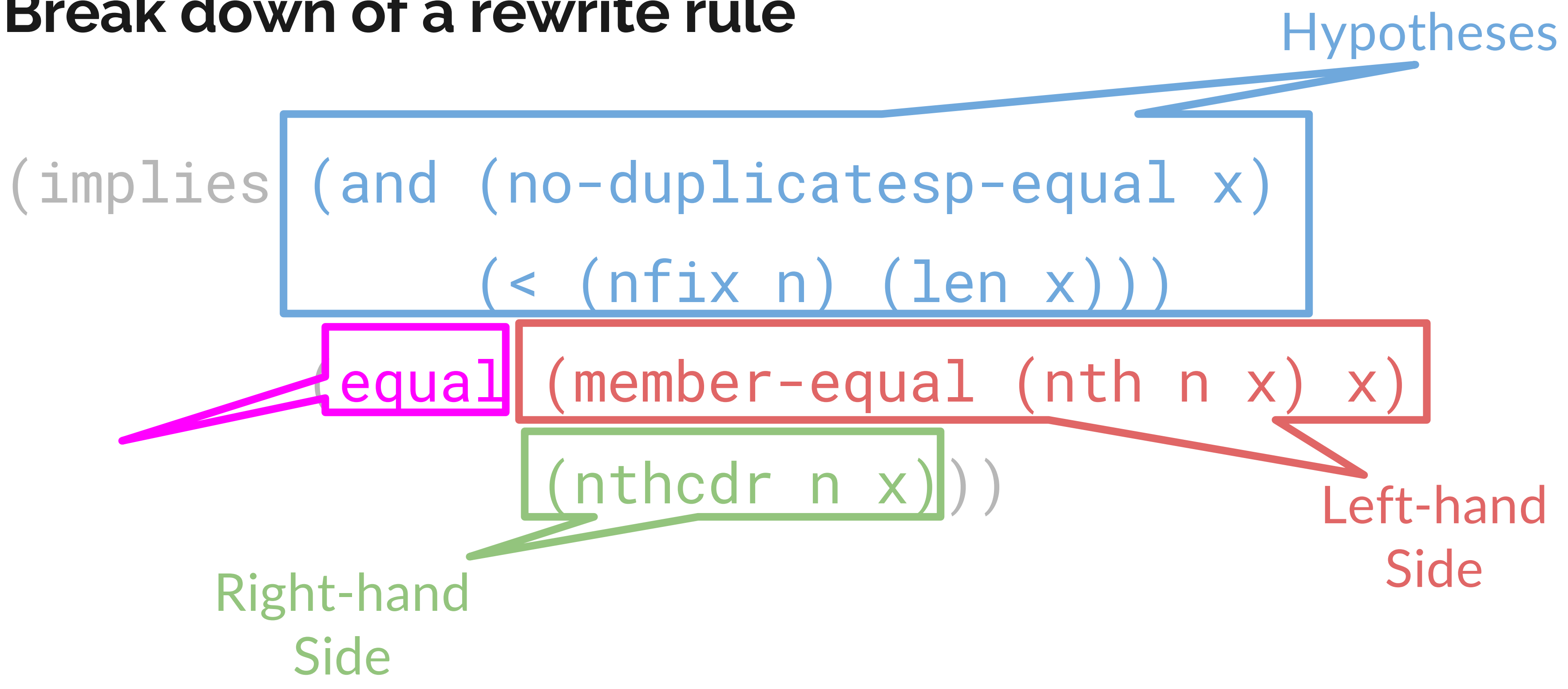
```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

Break down of a rewrite rule



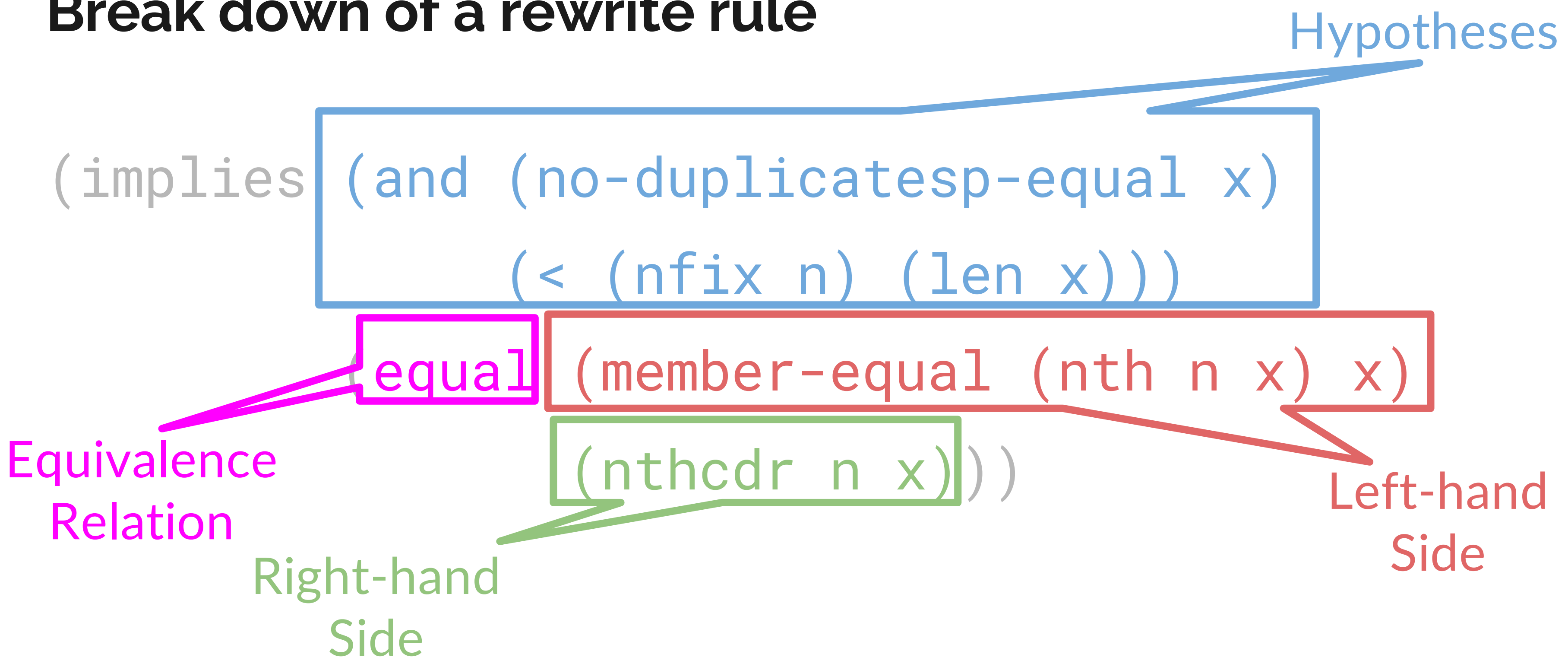
```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

Break down of a rewrite rule



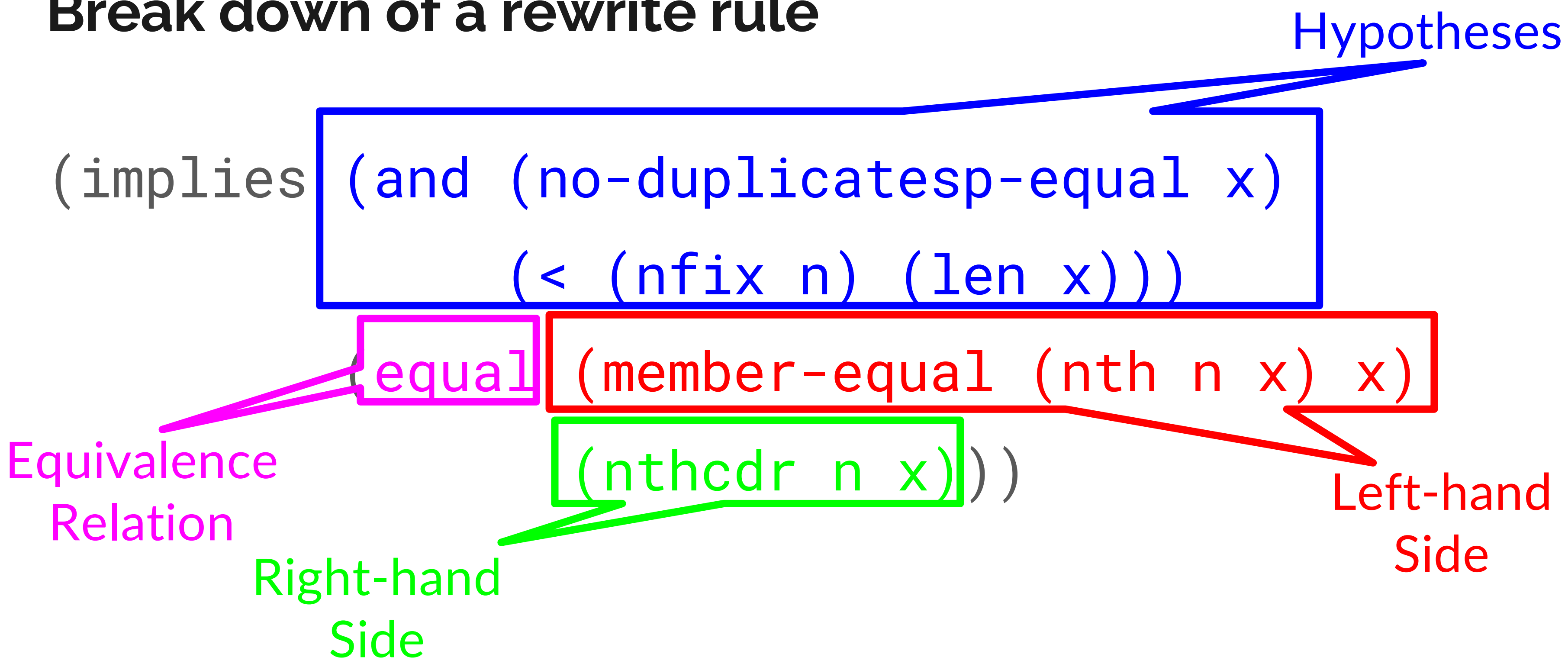
```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

Break down of a rewrite rule



```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

Break down of a rewrite rule



```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

Variant

```
(implies (< (nfix n) (len x))
         (member-equal (nth n x) x))
```

Variant

Hypothesis

```
(implies (< (nfix n) (len x))
         (member-equal (nth n x) x))
```

LHS

RHS = ?

Equiv relation = ?

Variant

Hypothesis

```
(implies (< (nfix n) (len x))
         (member-equal (nth n x) x))
```

LHS

RHS = T

Equiv relation = IFF

Applying Rewrite Rules

- A rewrite rule is a theorem with universally quantified variables — so we can instantiate it with any terms we want substituted for those variables.
- If we instantiate it with some substitution and can show that the hypotheses are true, it produces an equivalence (LHS \ subst) ≡ (RHS \ subst).
- If (LHS \ subst) occurs in our goal (and in a context where we can substitute based on the equivalence relation) then we can replace it with (RHS \ subst).

In practice

Go in the other direction: For some term in the theorem we want to prove, look for rewrite rules that might be able to simplify it.

In ACL2, rewrite rules are catalogued by leading function symbol of the LHS.

- Get the list of rewrite rules for the leading function symbol of our target term
- Try each of these rewrite rules until one can be applied successfully.

Rewrite Rule Operation

```
(implies (and (no-duplicatesp-equal x)
              (< (nfix n) (len x)))
         (equal (member-equal (nth n x) x)
                (nthcdr n x)))
```

To what terms can we apply this rewrite rule?

What terms match?

- `(nthcdr n x)`
- `(member-equal k x)`
- `(member-equal (nth a b) c)`
- `(member-equal (nth (nfix (index-of q x)) x) x)`
- `(member-equal (nth (foo) x) (true-list-fix x))`
- `(member-equal (nth (+ n m) (nthcdr n x)) (nthcdr n x))`

What terms match?

- ~~`(nthcdr n x)`~~ – unifies with RHS, not LHS (wrong leading function symbol)
- ~~`(member-equal k x)`~~ — first argument isn't a call of `nth`
- ~~`(member-equal (nth a b) c)`~~ — one occurrence of x matches b, one matches c
- `(member-equal (nth (nfix (index-of q x)) x) x)`
    - OK: substitution `((n . (nfix (index-of q x)))    (x . x))`
- ~~`(member-equal (nth (foo) x) (true-list-fix x))`~~ — one occurrence of x matches x, other matches `(true-list-fix x)`
- `(member-equal (nth (+ n m) (nthcdr n x)) (nthcdr n x))`
    - OK: substitution `((n . (+ n m))    (x . (nthcdr n x)))`

Rewriting a Term with a Rule — Summary

1. Check that the rule’s equivalence relation is appropriate to the context:
    a. OK to substitute with EQUAL always
    b. OK to substitute with IFF in Boolean contexts such as the first argument of IF
    c. User equivalence relations and congruence rules — advanced topic
2. UNIFY the target with the rule’s LHS to get a substitution such that target = LHS \ subst.
3. For each hypothesis:
    a. Rewrite it under the substitution
    b. If it can be reduced to T (“relieved”), then continue, else give up.
4. If successful, continue by rewriting the rule’s RHS under the substitution.

“Rewrite a term under a substitution”

Recur down the term’s structure:

- On a variable, return its substitution
- On a constant (quote), return itself
- On a function call, first recur over the arguments, then try to apply rewrite rules to the function applied to the resulting rewritten arguments. (“Inside out” rewriting)
    - Also can execute the function if args rewrite to constants
- On a lambda call, first recur over the arguments, then (if expanding the lambda) use them to form a new substitution and rewrite the body under that substitution

Deciding not to apply a rewrite

Always OK to decide not to apply a rule. “Syntaxp hyps” allow decision based on syntax of bound variables.

Example: how to simplify terms like `(+ 1 1 x)`?

If this were `(+ (+ 1 1) x)` it would reduce automatically, but instead we have `(+ 1 (+ 1 x))`.
So, rewrite `(+ a (+ b c))` to `(+ (+ a b) c)` — but only if they’re constants:

```
(implies (syntaxp (and (quotep a) (quotep b))
         (equal (+ a (+ b c)) (+ (+ a b) c)))
```

Free Variables

- So far, the substitution is determined only by unifying the target with the LHS
- A rewrite rule can contain variables in hyps or RHS that don’t occur in the LHS — then these aren’t assigned by the unifying substitution
- “We” can assign these however we want (to make the hyps true, to make the RHS nicer) — how to automate?
- ACL2 strategies:
    - Match hypothesis with free variable to known-true term
    - For hypotheses of the form (equal freevar (term with bound vars)) – bind freevar to result of rewriting term
    - BIND-FREE form lets rewrite rule author programmatically construct bindings for free variables

Example – Transitivity of `subsetp-equal`

```
(implies (and (subsetp-equal x y)
              (subsetp-equal y z))
         (subsetp-equal x z))
```

Suppose we’re trying to rewrite `(subsetp-equal a b)` so we get substitution `((x . a) (z . b))`

Come to the first hypothesis — has free variable `y` so in order to proceed we need a known-true term matching `(subsetp-equal a <something>)`

Unreliable rule because if such a term doesn’t exist, we’ll give up on trying to apply it

Often have two versions of this rule, only differing in which hyp comes first.

Example – Cancellation of common factors

```
(defthm divide-out-common-factors-equal
  (implies (and ...   ;; ???
                (acl2-numberp factor)
                (not (equal 0 factor))
                (acl2-numberp x)
                (acl2-numberp y))
           (iff (equal x y)
                (equal (* (/ factor) x) (* (/ factor) y)))))
```

Example – Cancellation of common factors

```
(defthm divide-out-common-factors-equal
  (implies (and (bind-free ...  ;; ???
                           (factor))
                (acl2-numberp factor)
                (not (equal 0 factor))
                (acl2-numberp x)
                (acl2-numberp y))
           (iff (equal x y)
                (equal (* (/ factor) x) (* (/ factor) y)))))
```

Example – Cancellation of common factors

```
(defthm divide-out-common-factors-equal
  (implies (and (bind-free ...
                 ;; Examine the syntactic forms of the bindings of x and y
                 ;; Determine if there is a common subterm multiplied with all the
                 ;; summands in x and y
                 ;; Bind factor to that subterm if so, else return NIL (don't apply this rule)
                           (factor))
                (acl2-numberp factor)
                (not (equal 0 factor))
                (acl2-numberp x)
                (acl2-numberp y))
           (iff (equal x y)
                (equal (* (/ factor) x) (* (/ factor) y)))))
```

Example – Cancellation of common factors

```
(defthm divide-out-common-factors-equal
  (implies (and (bind-free (and (or (not (equal x ''0))
                                    (case-match y (('binary-+ & &) t)))
                                (or (not (equal y ''0))
                                    (case-match x (('binary-+ & &) t)))
                                (let ((factors (find-common-factors-in-polys x y)))
                                  (and (consp factors)
                                       `((factor . ,(car factors))))))
                           (factor))
                (acl2-numberp factor)
                (not (equal 0 factor))
                (acl2-numberp x)
                (acl2-numberp y))
           (iff (equal x y)
                (equal (* (/ factor) x) (* (/ factor) y)))))
```

backup

```
(defun collect-factors (x)
  (case-match x
             (('quote &) nil)
             (('binary-* a b) (append (collect-factors a)
                                      (collect-factors b)))
             (& (list x))))

(mutual-recursion
 (defun find-common-factors-in-poly (x)
  (declare (xargs :mode :program))
  (case-match x
             (('binary-+ a b) (find-common-factors-in-polys a b))
             (('unary-- a) (find-common-factors-in-poly a))
             (('binary-* & &) (collect-factors x))
             (& (list x))))

 (defun find-common-factors-in-polys (x y)
   (cond ((equal x ''0) (find-common-factors-in-poly y))
             ((equal y ''0) (find-common-factors-in-poly x))
             (t (intersection-equal (find-common-factors-in-poly x)
                                      (find-common-factors-in-poly y))))))
```