Leibniz Axiom

```
(defun f (x)
  :ic (natp x)
  :oc (natp (f x))
  (+1 (f x)))
```

IC ⇒ OC

A1 $(natp\ x) \Rightarrow (natp\ (f\ x))$

IC ⇒ function = body

A2 $(natp\ x) \Rightarrow (f\ x) = (+1\ (f\ x))$

$\neg B$

T1 $(natp\ x) \Rightarrow (x \neq (x+1))$

$\equiv \{ T1 |_{((x\ (f\ x)))} \}$

$(natp\ (f\ x)) \Rightarrow ((f\ x) \neq ((f\ x)+1))$

Aside

$A \Rightarrow (B \Rightarrow C)$ (if A (if B C))

vs

$(A \Rightarrow B) \Rightarrow C$ (if (if A B) C)

$(in\ a\ (rest\ x)) \Rightarrow$ C

$\equiv \{ A1 \}$

B

$(natp\ x) \Rightarrow ((f\ x) \neq ((f\ x)+1))$

$\equiv \{$ Combine with A2 $\}$

$(natp\ x) \Rightarrow B \wedge \neg B$

$\equiv \{ p \wedge \neg p \equiv nil \}$

$(natp\ x) \Rightarrow nil$

Φ_{BS}

$\Phi_{BS}((x\ 4))$

$\equiv$

$t \Rightarrow nil$

$\equiv nil$

```
(defunc f (x)
  :ic (natp x)
  :oc (natp (f x))
  (+ x y))
```

Definitional Principle

The function f defined as

```
(defunc f (x1 ... xn)
  :input-contract ic
  :output-contract oc
  <body>)
```

is admissible provided

1) f is a new function symbol
2) The variable x_i are distinct

~~(defunc f (x x) call (f 2 3))~~

3) A body is a term possibly using f recursively as a function symbol mentioning no variables freely.

term = an expression that is well formed given the context of a history.

```
(defunc f (x)
  :ic (natp x)
  :oc (natp y)
  (cons (f (x)) y))))
```

Possibly bad

BAD

BAD

```
(defunc g (x)
  :ic (integerp x)
  :oc (integerp (g x))
  (+ 1 x))
```

natp

4) The function must terminate. This is the "non-trivial" obligation

5) IC ⇒ OC (output contract violation)

6) The body contracts hold under the assumption that ic holds. (BodyContractViolation)

gives us: IC ⇒ (f x) ≡ <body>

```
(defunc f (x)
  :ic (natp x)
  :oc (integerp (f x))
  (if (equal x 0)
      1
    (+1 (f (- x 1)))))
```

```
(defunc f (x y)
  :ic (and (natp x) (natp y))
  :oc (rationalp (f x y))
  (/ x y))
```

```
(defunc f (x y)
  :ic (and (natp x) (natp y))
  :oc (integerp (f x y))
  (cond ((equal x 0) 1)
        ((< x 0) (f -y -1))
        (t (+1 (f (- x 1) y)))))
```

A function m is a measure of f if:

1) m has the same ic as f
2) the OC for m is natp
3) m has the same parameters as f and is admissible
4) For each recursive call in f, given the conditions that lead to recursion m must decrease with the change in parameters.

$$(natp\ x) \wedge (listp\ y) \wedge \neg(endp\ y)$$
$$\wedge (x \neq 0) \Rightarrow (m\ x\ y) > (m\ (-x\ 1)\ (cons\ x\ y))$$

```
(defunc f (x y)
  ;ic (and (natp x) (listp y))
  ;oc (listp (f x y))
  (cond ((endp y) y)
        ((equal x 0) y)
        (t (f (- x 1) (cons x y)))))
```

```
(defunc m (x y)
  :ic (and (natp x) (listp y))
  :oc (natp (m x y))
  (* x 100))
```

```
(defunc f (x y)
  :ic (and (natp x)(natp y))
  :oc (integerp (f x y))
 (cond ((equal x 0) 1)
       ((< x 0) (f x y))
       (t (f 1 (f (- x 1) y)))))
```

Obligation 1

false

$(natp\ x) \wedge (natp\ y) \wedge (x \neq 0) \wedge (x < 0) \Rightarrow ((m\ x\ y) > (m\ x\ y))$

Obligation 2

$(natp\ x) \wedge (natp\ y) \wedge (x \neq 0) \Rightarrow ((m\ x\ y) > (m\ (-x\ 1)\ y))$

C1. $(natp\ x)$

C2 $(natp\ y)$

C3. $(x \neq 0)$

$(m\ x\ y)$

$\equiv$ {Def m}

x

$> \{Arithmetic\}$

$(-x\ 1)$

$\equiv$ {Def m} ← C1, C3 ...

$(m\ (-x\ 1)\ y)$

$\Rightarrow (m\ x\ y) = x > (-x) \equiv (m\ (-x)\ y)$

```
(defunc f (x_1 ... x_i ... x_n)
  :ic (and (...) (listp x_i) ...)
  :oc (... (f x_1 ... x_i ... x_n))
 (cond ((endp x_i) <stop>
        ((...) (f ... (rest x) ...))
        ((...) (f ... (rest x) ...))))

(defunc m (x_1 ... x_i ... x_n)
   :ic (and (...) (listp x_i) ...)
   :oc (natp (f x_1 ... x_i ... x_n))
   (len x_i))
```

```
(defunc f (x_1 ... x_i ... x_n)
  :ic (and (...) (natp x_i) ...)
  :oc (... (f x_1 ... x_i ... x_n))
  (cond ((equal x_i 0) <stop>)
        ((...) (f ... (- x_i 1) ...))
        ((...) (f ... (- x_i 1) ...))))

(defunc m (x_1 ... x_i ... x_n)
  :ic (and (...) (natp x_i) ...)
  :oc (natp (f x_1 ... x_i ... x_n))
  x)
```

```
(defunc f (r)
  :ic (rationalp r)
  :oc (rationalp (f r))
  (if (<= r 0)
      0
      (f (/ r 2))))

(defunc f (i)
  :ic (integerp i)
  :oc (integerp (f i))
  (if (<= i -5)
      i
      (f (- i 1))))
```

```
(defunc m (r)
  :ic (rationalp r)
  :oc (natp (m r))
  r)

(defunc m (i)
  :ic (integerp i)
  :oc (natp (m i))
  (* (+ i 5) (+ i 5)))
```

```
(defunc f (i)
  :ic (integerp i)
  :oc (integerp (f i))
  (if (> i 1000)
      0
      (f (+ i 1))))

(defunc m (i)
  :ic (integerp i)
  :oc (natp (m i))
  (if (> i 1000)
      0
      (- 1000 i)))
```

```
(defunc f (i l)
  :ic (and (integerp i) (listp l))
  :oc (natp (f i l))
  (cond ((and (equal i 0) (endp l)) 0)
        ((equal i 0) (f i (rest l)))
        ((< i 0)     (f (+ i 1) l))
        (t           (f (- i 1) l))))
```

```
(defunc m (i l)
  :ic (and (integerp i) (listp l))
  :oc (natp (m i l))
  (+ (len l) (if (< i 0) (- i) i))
     or
  (+ (* i i) (len l))
     or
  (+ i (len l))
     or
  (+ (abs i) (len l))
```

Ob 1

$(\text{integerp } i) \wedge (\text{listp } l) \wedge \neg((i=0) \wedge (\text{endp } l)) \wedge (i=0) \Rightarrow (m\ i\ l) > (m\ i\ (\text{rest } l))$

Ob 3

$(\text{integerp } i) \wedge (\text{listp } l) \wedge \neg((i=0) \wedge (\text{endp } l)) \wedge (i \neq 0) \wedge (i < 0) \Rightarrow (m\ i\ l) > (m\ (-i\ 1)\ l)$

① Turing's Indecidability Problem.
② It justifies recursion.
③ It justifies the design recipe
④ It justifies generative recursion
⑤ It justifies induction

```
(defunc nind (n)
  :ic (natp n)
  :oc (natp (nind n))
  (if (equal n 0)
      0
      (nind (- n 1))))
```

```
(defunc sum-n (n)
  :ic (natp n)
  :oc (natp (sum-n n))
  (if (equal n 0)
      0
      (+ n (sum-n (- n 1)))))
```

Assume φ_{n-1} Prove φ_n

$\varphi_{n-1} \Rightarrow \varphi_n$

$\varphi_0 \Rightarrow \varphi_1 \Rightarrow \varphi_2 \ldots \varphi_{n-1} \Rightarrow \varphi_n$

..

Induction Scheme for nind
(Is that nind gives rise to)

① $\neg(\text{natp } n) \Rightarrow \varphi$
② $(\text{natp } n) \wedge (n = 0) \Rightarrow \varphi$
③ $(\text{natp } n) \wedge (n \neq 0) \wedge \varphi_{(n\ (-\ n\ 1))} \Rightarrow \varphi$

φ_{app_assoc} $(listp\ x) \wedge (listp\ y) \wedge (listp\ z) \Rightarrow ((app\ (app\ x\ y)\ z) = (app\ x\ (app\ y\ z)))$

$\varphi_{sum\,n}$ $(natp\ n) \Rightarrow ((sum\text{-}n\ n) = (n * (n+1))/2)$

Obligation 1

$\neg(natp\ n) \Rightarrow ((natp\ n) \Rightarrow ((sum\text{-}n\ n) = (n * (n+1))/2))$ φ

Obligation 2

$(natp\ n) \wedge (n = 0) \Rightarrow ((natp\ n) \Rightarrow ((sum\ n\ n) = (n * (n+1))/2))$ φ

Obligation 3

$(natp\ n) \wedge (n \neq 0) \wedge [(natp\ (-\ n\ 1)) \Rightarrow ((sum\text{-}n\ (-\ n\ 1)) = (/\ (*\ (-\ n\ 1)\ (+\ (-\ n\ 1)\ 1))\ 2))]$

$\Rightarrow ((natp\ n) \Rightarrow ((sum\ n) = (n * (n+1))/2))$

C1. ¬(natp n)

C2. (natp n)

C3. nil {C1, C2}

nil => φ ≡ true

If $\varphi_{app.assoc}$

C1. ¬(natp x)

C2. (listp x)

C3. (listp y)

C4. (listp z)