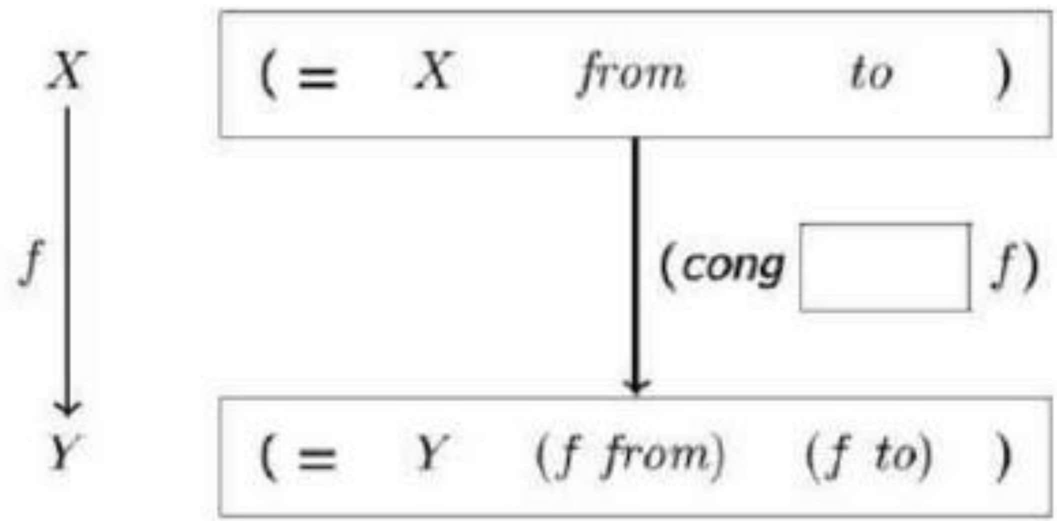
cong



Leibnitz Law

If two expressions are equal, then whatever is true for one is true for the other.

replace

The Law of replace

If *target* **is an**

  (= X *from to*),

mot **is an**

  ($\to$ X
    $\mathcal{U}$),

and *base* **is a**

  (*mot from*)

then

  (replace *target*
    mot
    base)

is a

  (*mot to*).

Difference: `twice` vs `double`

Beispiel n+1 verdoppeln

Twice:

```
(+ (add1 n) (add1 n))
```

Double:

```
(add1 (add1 (+ n n)))
```

Difference: `twice` vs `double`

Beispiel n+1 verdoppeln

Twice:

(+ (add1 n) (add1 n))

Double:

(add1 (add1 (+ n n)))

Beweis für

Even though

(+ *n* (add1 *j*))

is not the same Nat as

(add1 (+ *n* *j*)),

they are equal Nats.

`(add1 (+ n-1 n-1))`

equals

`(+ n-1 (add1 n-1))`

```
(define step-twice=double
  (λ (n-1)
    (λ (twice=double_n-1)
      [          ])))
```

Typ der Box:

```
(= Nat
  (twice (add1 n-1))
  (double (add1 n-1)))
```

Der Typ entspricht dem Typ:

```
(= Nat
  (add1
    (+ n-1 (add1 n-1)))
  (add1
    (add1 (double n-1))))
```

von

```
(cong twice=double_n-1
  (+ 2))
```

Ist der Typ

```
(= Nat
  (add1
    (add1 (+ n-1 n-1)))
  (add1
    (add1 (double n-1))))
```

Erinnerung an
`add1+=+add1`

Vergleich zu was wir wollen:

```
(= Nat
  (add1
    (+ n-1 (add1 n-1)))
  (add1
    (add1 (double n-1))))
```

`replace` kann `add1` nach innen verschieben

von

```
(cong twice=double_n-1
  (+ 2))
```

Ist der Typ

```
(= Nat
  (add1
    (add1 (+ n-1 n-1)))
  (add1
    (add1 (double n-1))))
```

Vergleich zu was wir wollen:

```
(= Nat
  (add1
    (+ n-1 (add1 n-1)))
  (add1
    (add1 (double n-1))))
```

```
(double-Vec Atom 3
  (vec:: 'chocolate-chip
    (vec:: 'oatmeal-raisin
      (vec:: 'vanilla-wafer
        vecnil))))
```

is

```
(vec:: 'chocolate-chip
  (vec:: 'chocolate-chip
    (vec:: 'oatmeal-raisin
      (vec:: 'oatmeal-raisin
        (vec:: 'vanilla-wafer
          (vec:: 'vanilla-wafer
            vecnil)))))).
```

Solve Easy Problems First

If two functions produce equal results, then use the easier one when defining a dependent function, and then use replace to give it the desired type.

The Law of symm

If e **is an** $(= X\ \textit{from}\ \textit{to})$**, then** (symm e) **is an** $(= X\ \textit{to}\ \textit{from})$**.**

Kapitel 10

The Law of Σ

The expression

```
(Σ ((x A))
   D)
```

is a type when A **is a type, and** D **is a type if** x **is an** A**.**

The Commandment of cons

If p **is a**

```
(Σ ((x A))
  D),
```

then p **is the same as**

```
(cons (car p) (cdr p)).
```

Data constructor	Typ constructor	Typ constructor
Cons	Pair	Σ
Lambda	->	Π

The Law of Σ

The expression

$$(\Sigma\ ((x\ A))\ D)$$

is a type when A is a type, and D is a type if x is an A.

Ist das ein Typ?

$$(\Sigma\ ((A\ \mathcal{U}))\ A)$$

Was sind Beispiel-Ausdrücke?

The Law of Σ

The expression

```
(Σ ((x A))
  D)
```

is a type when A is a type, and D is a type if x is an A.

Ist das ein Typ?

```
(Σ ((ℓ Nat))
  (Vec Atom ℓ))
```

Was sind Beispiel-Ausdrücke?

```
(cons 2
  (vec:: 'toast-and-jam
    (vec:: 'tea vecnil)))
```

$\boldsymbol{\Sigma} - \textbf{Ausdruck}$

Teilmenge der Nats / Prädikate

- $\boldsymbol{\Sigma}$ ((n Nat)) (Prim n)

Nicht Teilmenge der Nats

- $\boldsymbol{\Sigma}$ ((n Nat)) (Vec n Atom)

$Pair$ read as statement

(Pair A D)	

Σ read as statement

Σ ((n Nat)) (Prim n)	

Proofs for Σ

Expression	Proof
(Σ ((*es* (List Atom))) (= (List Atom) *es* (***reverse*** Atom *es*)))	