Verarbeitung unendlicher Datenstrukturen

Jetzt können wir z.B. die unendliche Liste aller geraden Zahlen oder aller Quadratzahlen berechnen:

```
- take 10 (filter (fn x => x mod 2=0) nat);
val it = [0,2,4,6,8,10,12,14,16,18] : int list


- take 10 (map (fn x => x*x) nat);
val it = [0,1,4,9,16,25,36,49,64,81] : int list
```

Unendliche Datenstrukturen

So können wir auch die Liste aller Primzahlen berechnen (Sieb des Eratosthenes):

```
fun all_primes () =
 let
   fun sieve (Stream (n,ns)) =
     Stream(n,
            fn()=>sieve (filter (fn x => x mod n<>0) (ns())))
           )
 in
   sieve (generateNat 2)
 end;

- take 10 (all_primes());
val it = [2,3,5,7,11,13,17,19,23,29] : int list
```

Unendliche Datenstrukturen

```
take 200 (all_primes());
val it =
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,
 103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,
 199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,
 313,317,331,337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,
 433,439,443,449,457,461,463,467,479,487,491,499,503,509,521,523,541,547,557,
 563,569,571,577,587,593,599,601,607,613,617,619,631,641,643,647,653,659,661,
 673,677,683,691,701,709,719,727,733,739,743,751,757,761,769,773,787,797,809,
 811,821,823,827,829,839,853,857,859,863,877,881,883,887,907,911,919,929,937,
 941,947,953,967,971,977,983,991,997,1009,1013,1019,1021,1031,1033,1039,1049,
 1051,1061,1063,1069,1087,1091,1093,1097,1103,1109,1117,1123,1129,1151,1153,
 1163,1171,1181,1187,1193,1201,1213,1217,1223] : int list
```

4.9 Typ-Inferenz

Der SML-Compiler erlaubt Typ-Information auszulassen, wenn diese aus dem Kontext herleitbar (inferierbar) ist

$\Longrightarrow$ Typ-Inferenz

- Der Compiler leitet den allgemeinsten (d.h. möglichst polymorphen) Typ her. Z.B. hat der Wert `nil` u.a. die Typen

```
                int list
(bool * string list) list
    ({x: int; y: 'a} list
```

Der allgemeinste Typ ist aber

```
'a list
```

Typ-Ausdrücke

- Typ-Ausdruck = Ausdruck bestehend aus Typ-Konstanten (Basis-Typen), Anwendungen von Typ-Operatoren und Typ-Variablen, die unbekannte, beliebige Typ-Ausdrücke repräsentieren:

▷ `string -> int`

▷ `int * (unit -> int list)`

▷ `'a -> 'a`

▷ `int * 'a`

▷ `'a list * 'b list -> 'a * 'b list`

Typ-Instanzen

- Durch Einsetzen eines Typs für eine Typ-Variable erhält man eine Typ-Instanz:

▷ `int -> int` ist eine Instanz von `'a -> 'a`

▷ `int -> string` ist keine Instanz von `'a -> 'a`

▷ `(int * int) list` ist eine Instanz von `('a * 'b) list`

▷ `(int * ((int -> string) list)) list` ist eine Instanz von `('a * 'b) list`

4.9.1 Typ-Annotationen

Wenn dem Programmierer der hergeleitete Typ eines Ausdrucks zu allgemein ist, kann er ihn mithilfe von Typ-Annotationen einschränken.

```
NONE;
val it = NONE : 'a option
NONE : int option;
val it = NONE : int option
fun f x = [x];
val f = fn : 'a -> 'a list
fun f x = [x] :int list;
val f = fn : int -> int list
fun f (x :int) = [x];
val f = fn : int -> int list
```

Typ-Annotationen

Der angegebene Typ muss eine Instanz des hergeleiteten Typs sein:

```
[NONE] : 'a list;
stdIn:17.1-17.17 Error: expression doesn't match constraint
  expression: 'Z option list
  constraint: 'a list
in expression: NONE :: nil: 'a list
fun f x = (x,x) : 'a * 'b;
stdIn:2.6-2.20 Error: expression doesn't match constraint
    expression: 'a * 'a
    constraint: 'a * 'b
in expression: (x,x): 'a * 'b
```

4.10 Inferenz-Regeln

Typinferenz-Regeln: Beispiel

... besagen wie der Typ eines Ausdrucks aus den Typen seiner Teilausdrüke hergeleitet wird.

Beispiel: Wenn x und y vom Typ `int` sind, dann ist $x + y$ auch vom Typ `int` (und umgekehrt). Schreibweise:

$$\frac{x : \text{int} \qquad y : \text{int}}{x{+}y : \text{int}}$$

- Die Anwendung der entsprechenden Inferenz-Regel für einen Ausdruck bestimmt eine Menge von Gleichungen (*constraints*), die die Beziehungen zwischen den Typen der Teilausdrücke beschreibt.
- Für einen Ausdruck e ergibt sich ein Gleichungssystem S.
  - ▷ Hat S eine Lösung $\Longrightarrow$ die Auswertung von e führt nie zu einem Typ-Fehler
  - ▷ Hat S keine Lösung $\Longrightarrow$ die Auswertung von e könnte fehlschlagen

Beispiel: Typ-Inferenz für `a+1`

$$\frac{x : \text{int} \qquad y : \text{int}}{x+y : \text{int}} \qquad \frac{\texttt{a} : \texttt{'a} \qquad \texttt{1} : \texttt{int}}{\texttt{a+1} : \texttt{'b}}$$

Durch *Unifikation* der Hypothesen und der Schlussfolgerungen$\Longrightarrow$
Gleichungssystem mit den Typ-Variablen `'a` und `'b`:

$$\left\{\begin{array}{l}\texttt{'a = int} \\ \texttt{int=int} \\ \texttt{'b = int}\end{array}\right. \Longleftrightarrow \left\{\begin{array}{l}\texttt{'a = int} \\ \texttt{'b = int}\end{array}\right. \Rightarrow \texttt{a : int} \text{ und } \texttt{a+1 : int}.$$

Typinferenz: Allgemeine Regeln

Case: $$\frac{e \texttt{ : 'a} \quad p_1 \texttt{ : 'a ... } p_n \texttt{ : 'a} \qquad e_1 \texttt{ : 'b ... } e_n \texttt{ : 'b}}{\texttt{case } e \texttt{ of } p_1 \texttt{ => } e_1 \texttt{ | ...| } p_n \texttt{ => } e_n \texttt{ : 'b}}$$

If: $$\frac{p \texttt{ : bool} \qquad A \texttt{ : 'a} \qquad B \texttt{ : 'a}}{\texttt{if } p \texttt{ then } A \texttt{ else } B \texttt{ : 'a}}$$

Anwendung: $$\frac{f \texttt{ : 'a} \mapsto \texttt{'b} \qquad a \texttt{ : 'a}}{f \ a \texttt{ : 'b}}$$

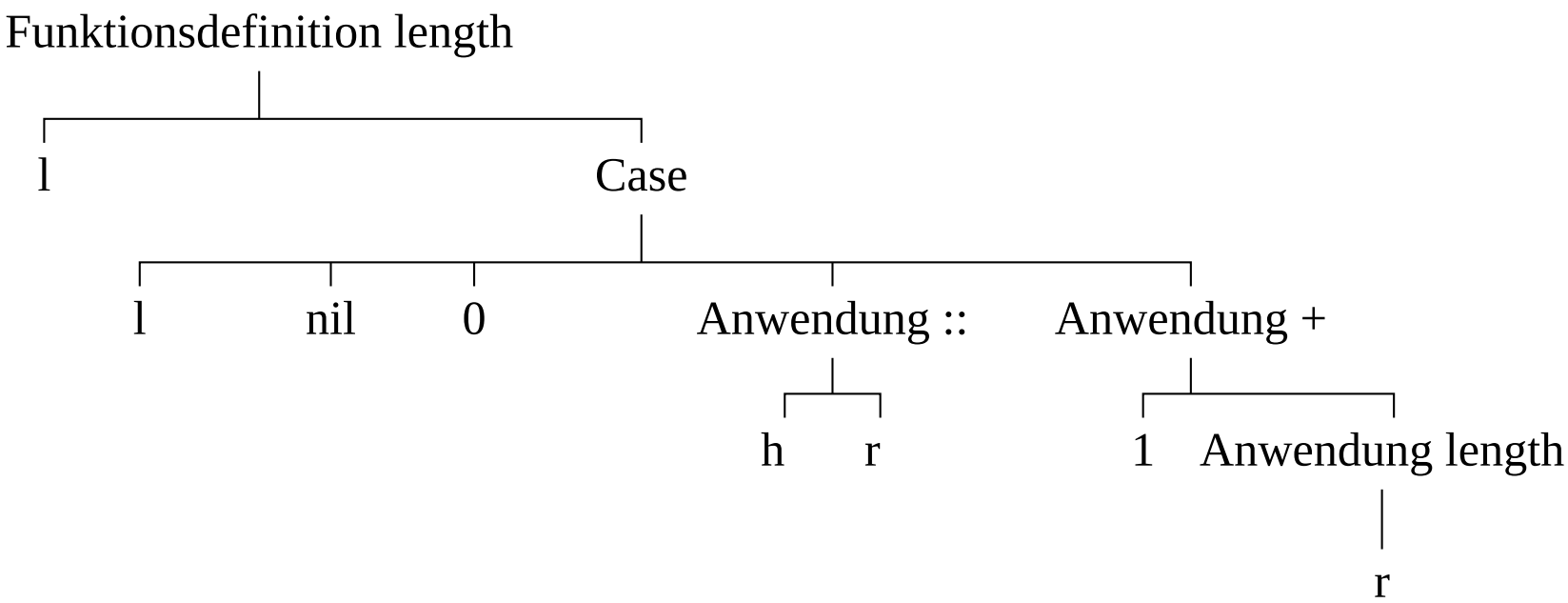
Funktionsdefinition: $$\frac{x \texttt{ : 'a} \qquad e \texttt{ : 'b}}{\texttt{fun } x \texttt{ = } e \texttt{ : 'a} \mapsto \texttt{'b}}$$

Typ-Inferenz

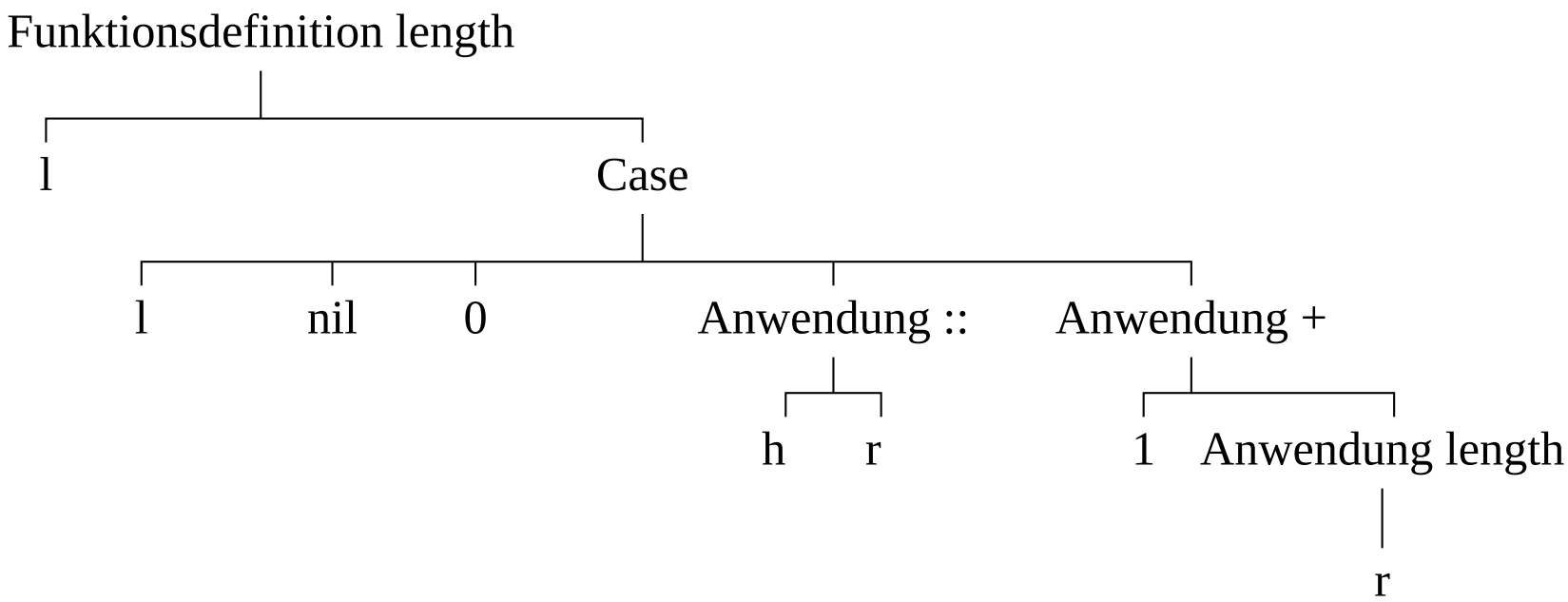
```
fun length l = case l of nil => 0
                       | h::r => 1 + length r
```

Typ-Inferenz

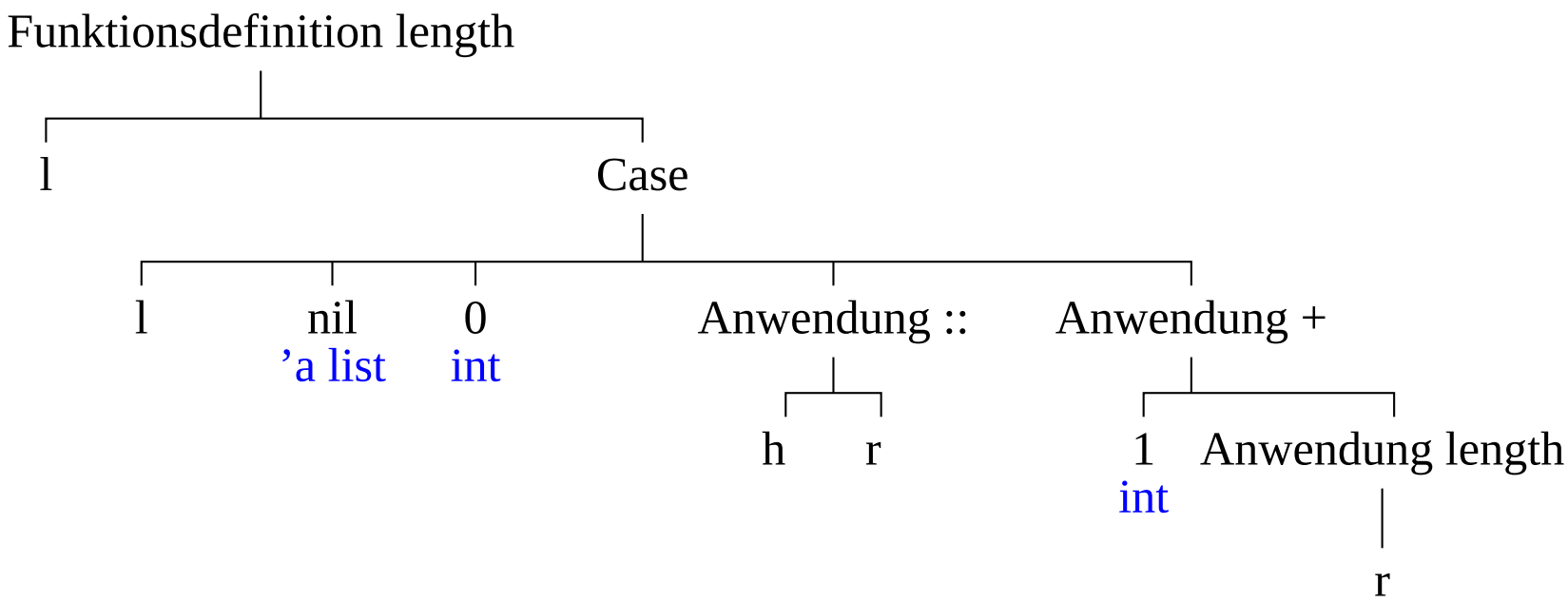
```
fun length l = case l of nil => 0
                       | h::r => 1 + length r
```



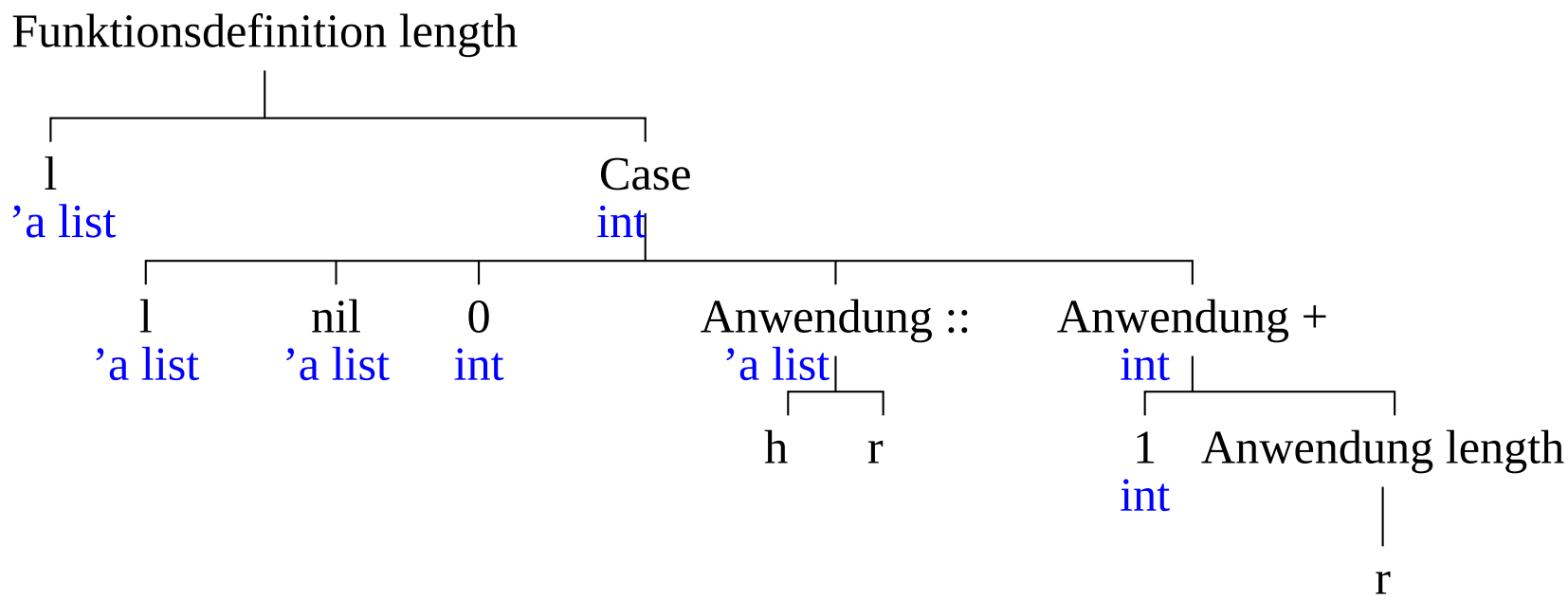
Typ-Inferenz



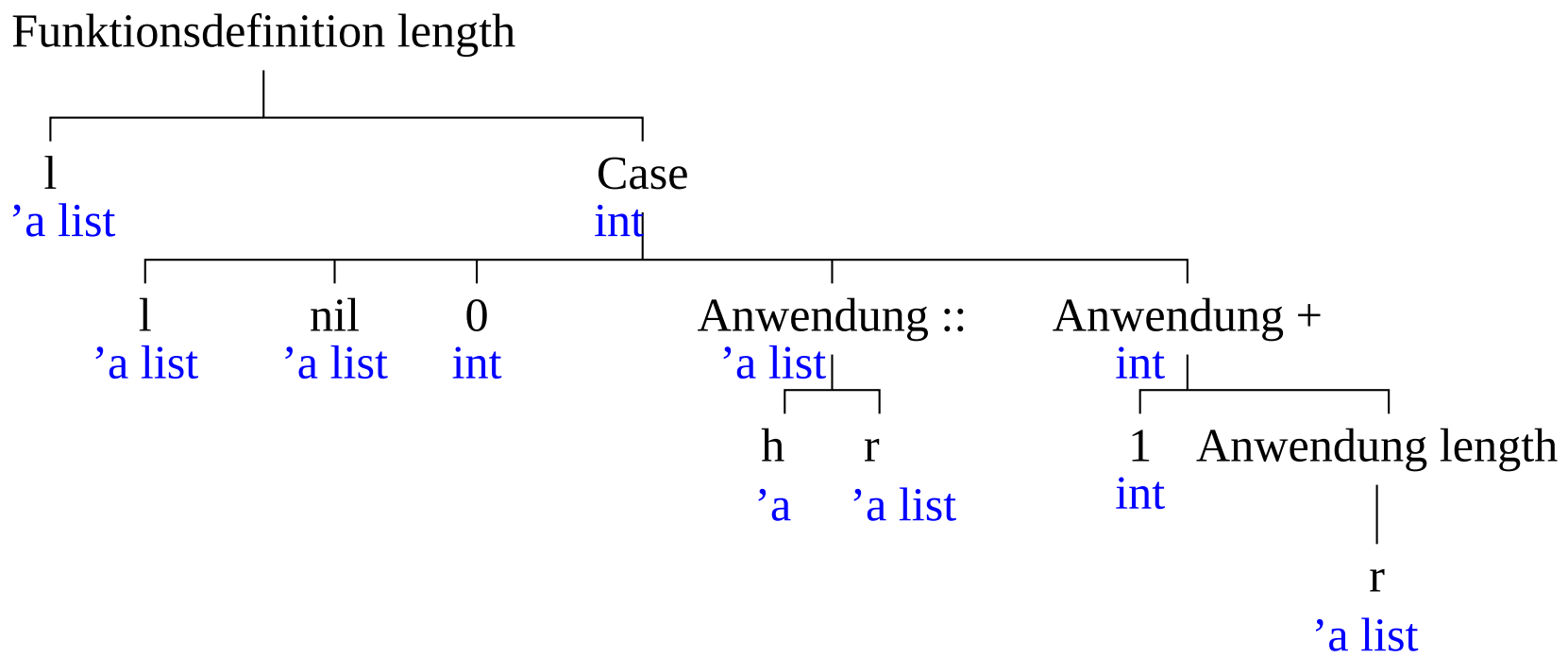
Konstanten


Funktionsdefinition length
l
Case
l
nil
'a list
0
int
Anwendung ::
h
r
Anwendung +
1
int
Anwendung length
r

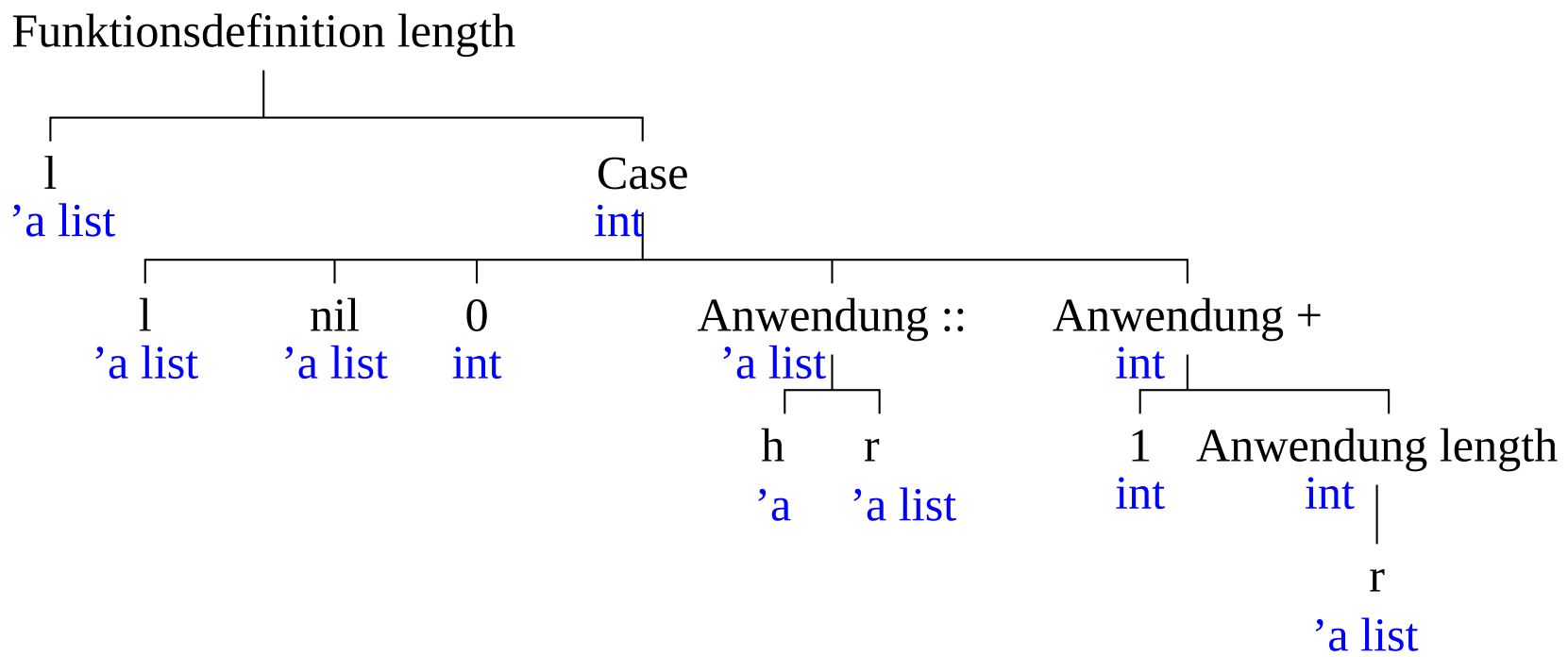
Case


Funktionsdefinition length
l
'a list
Case
int
l
'a list
nil
'a list
0
int
Anwendung ::
'a list
h
r
Anwendung +
int
1
int
Anwendung length
r

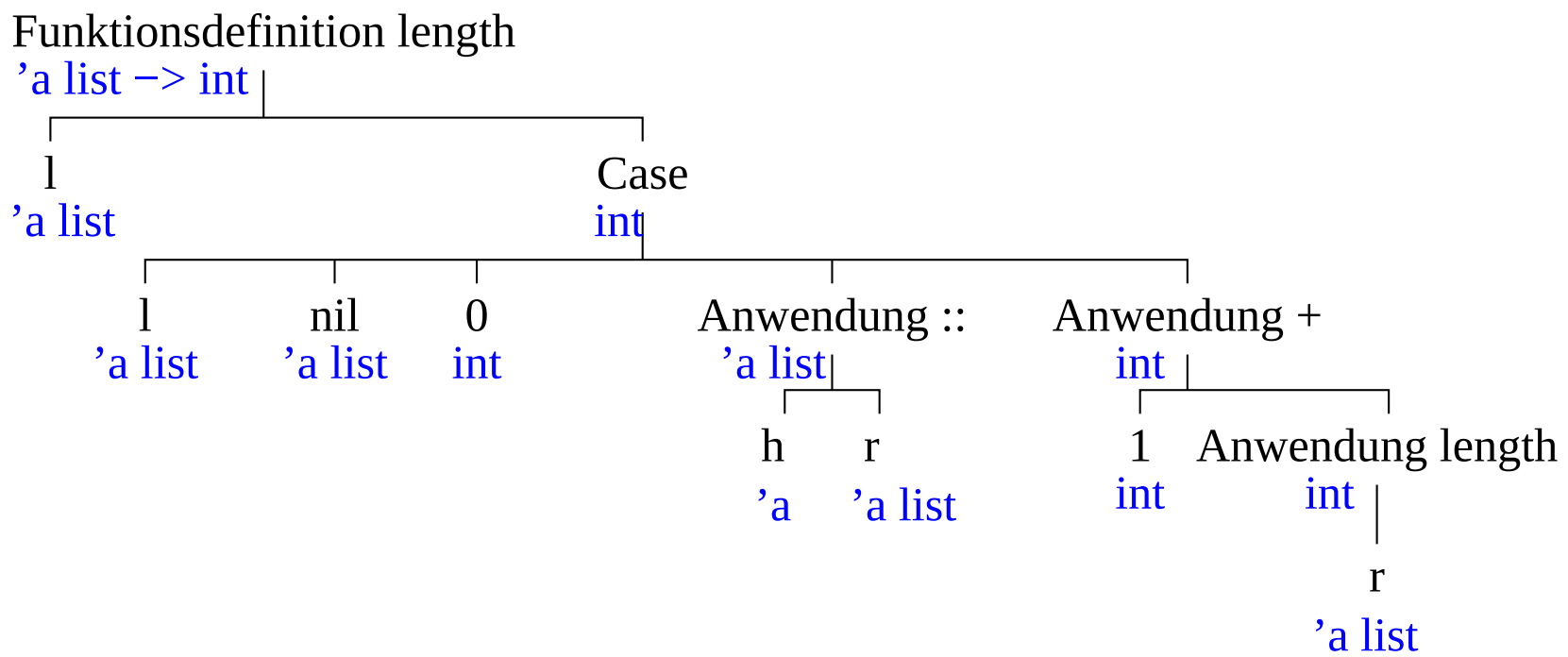
Anwendung des Konstruktors : :



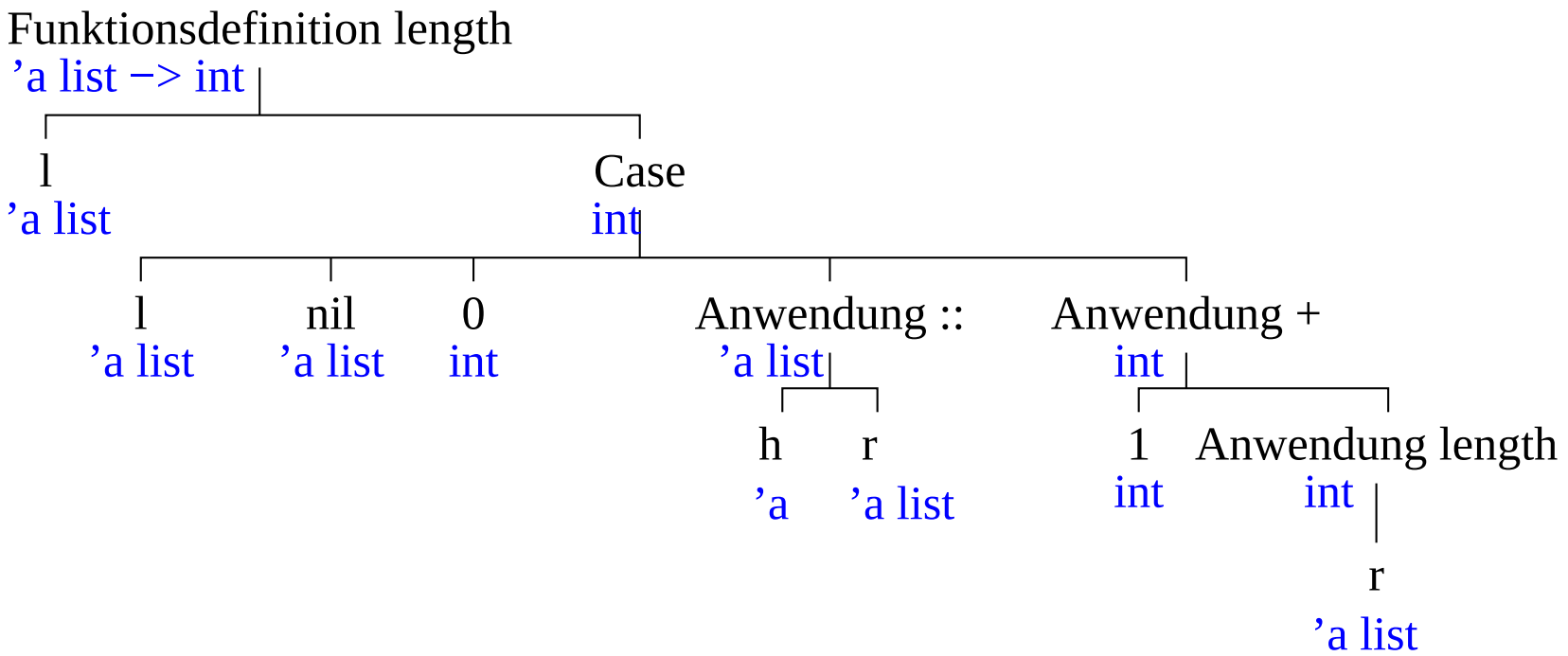
Anwendung der Addition



Anwendung der Funktion `length`



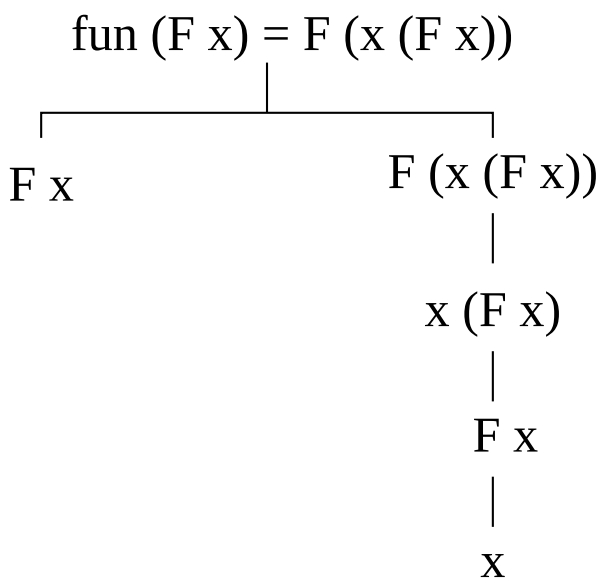
Funktionsdefinition `length`



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

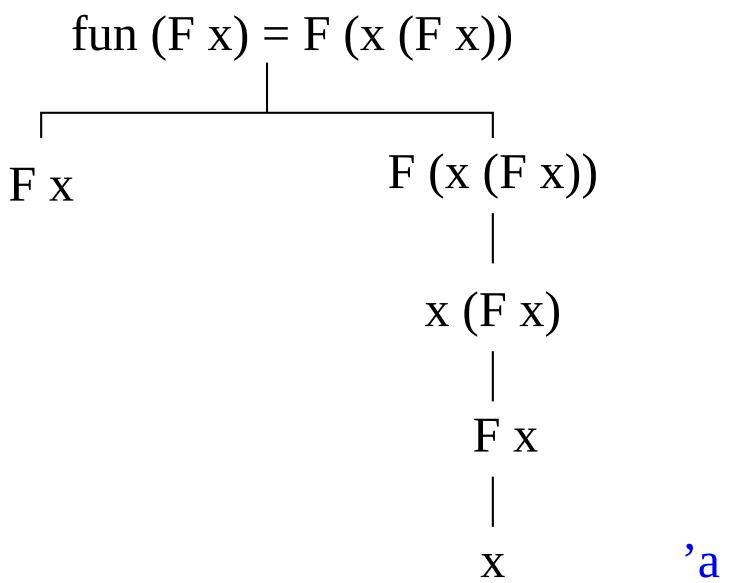
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

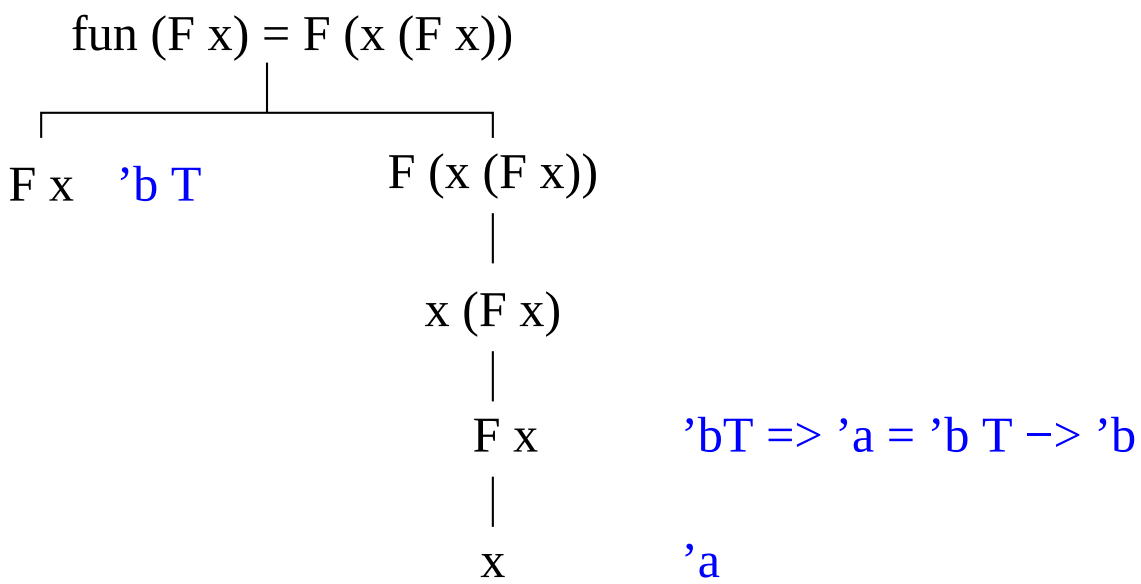
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

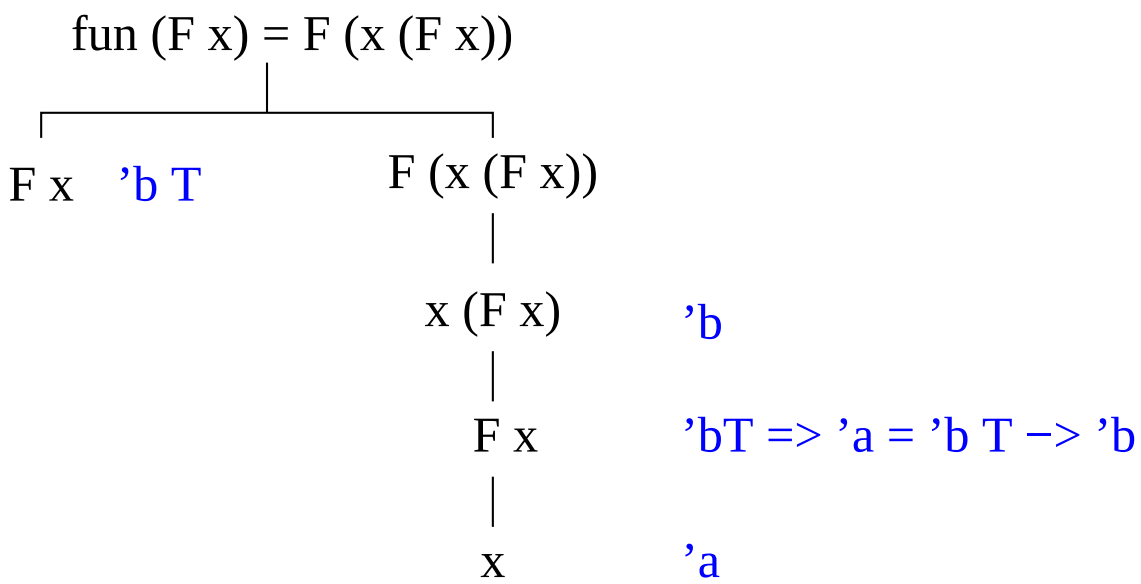
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

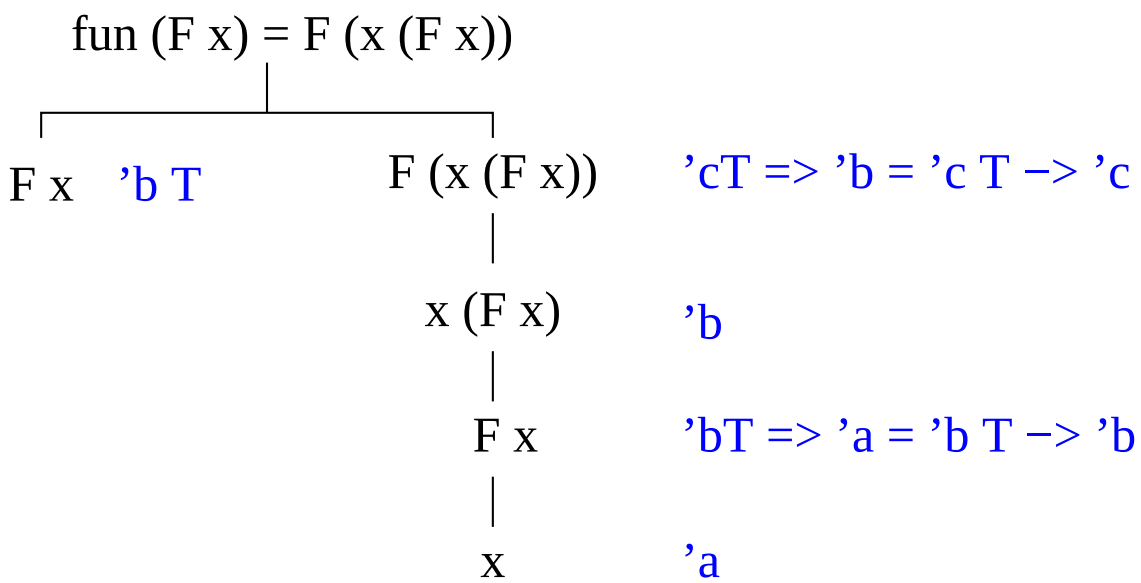
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

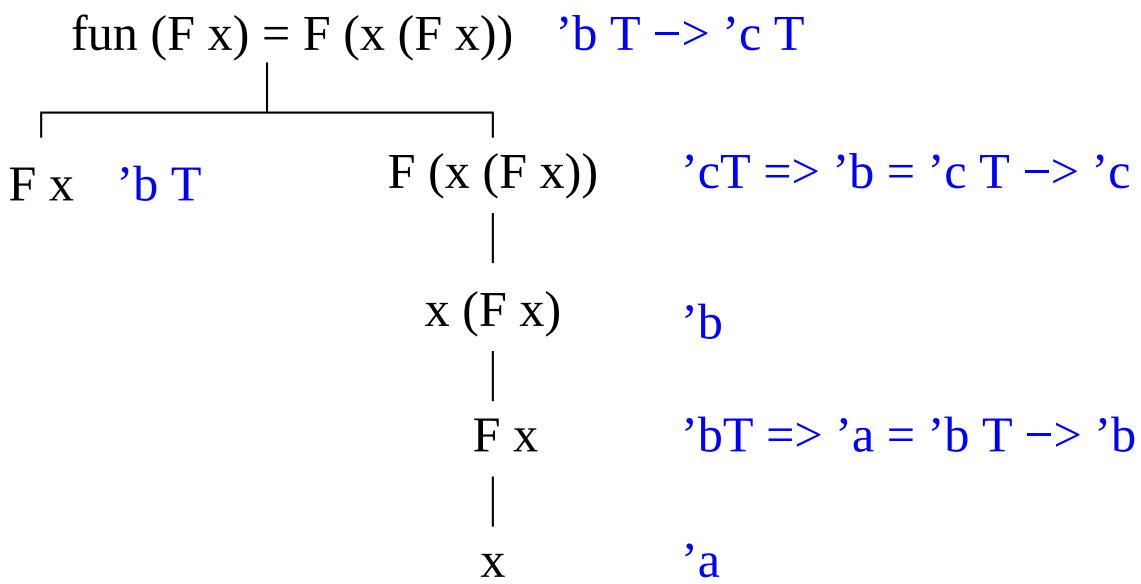
Welchen Typ hat `fun f (F x) = F (x (F x))`?



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

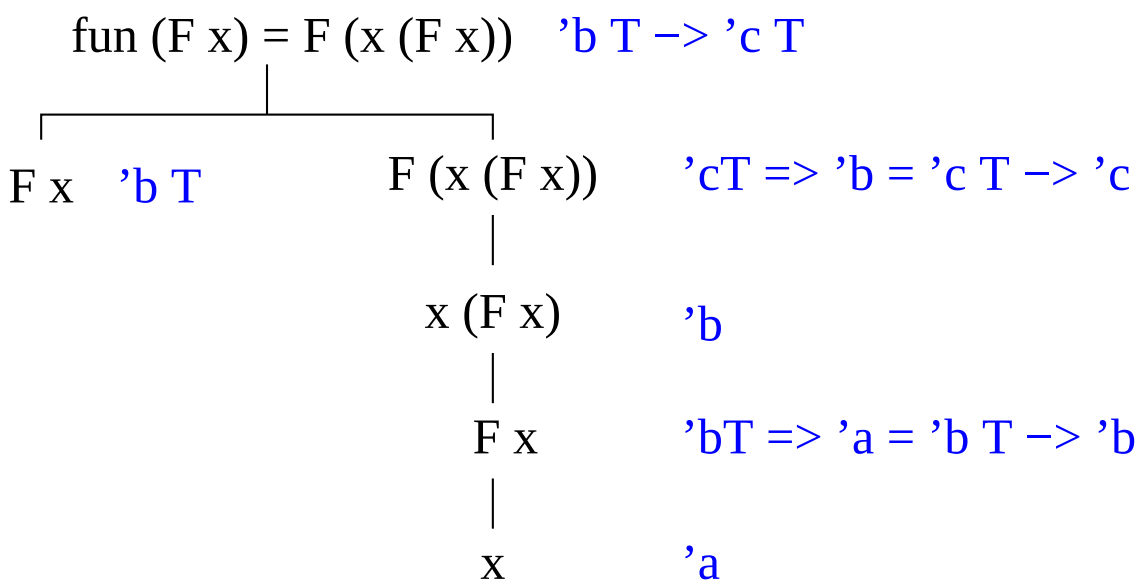
$\Longrightarrow$ `fun f (F x) = F (x (F x)) : 'b T -> 'c T`



Typ-Inferenz: Beispiel

Wir definieren: `datatype 'a T = F of 'a T -> 'a`

$\Longrightarrow$ `fun f (F x) = F (x (F x)) : ('c T -> 'c) T -> 'c T`



Typ-Inferenz: Beispiel

```
fun f (F x) = F (x (F x)) : ('cT -> 'c) T -> 'c T
```

Durch Umbennen der Typ-Variable `'c`:

```
fun f (F x) = F (x (F x)) : ('a T -> 'a) T -> 'a T
```

```
datatype 'a T = F of 'a T -> 'a
fun f (F x) = F (x (F x))
val f = fn : ('a T -> 'a) T -> 'a T
```

4.11 Ausnahmen

Ausnahmen sind Werte eines vordefinierten Typs `exn` $\in MT$.

Konstruktoren für die Werte des Typen `exn` können vom Benutzer definiert werden:

`exception` *AusnahmeKonstruktor* `[of` *Typ*`]`

Beispiel:

```
exception LeereListe
exception BannedWords of string list
```

Dadurch wurde der `exn`-Datentyp um zwei Exceptions erweitert. Das geht mit keinem anderen Datentyp!

Vordefinierte Ausnahmekonstruktoren

`Div`	bei Division durch Null
`Empty`	bei Zugriff auf eine leere Liste (`hd []`)
`Match`	bei unvollständigem Match in einem `case`-Ausdruck oder im Funktionskopf
`Fail`	ohne bestimmte Bedeutung, zur Benutzung durch den Programmierer

```
- 1 div 0;
uncaught exception divide by zero raised at: <stdin>
- tl (tl [1]);
uncaught exception Empty raised at: boot/list.sml:37.38-37.43
```

Der Typ `exn`

Der Typ **`exn`** ist das selbe unabhängig vom Typ des eingebetten Wertes.

```
LeereListe;
val it = LeereListe(-) : exn
- BannedWords ["viagra","rolex","medication"];
val it = BannedWords(-) : exn
```

$\Longrightarrow$ **`exn`** ist kein polymoprher Typ.

Ausnahmenverarbeitung

- Ausnahmen Werfen:
  - Eine Ausnahme `ex` kann bei der Laufzeit während einer Ausdrucksauswertung *geworfen* werden.
  - `ex` reist in die *Vergangenheit* zu den noch nicht zu Ende ausgewerteten Ausdrucksauswertungen.
- Ausnahmen Behandeln:

  Eine Ausnahme `ex`, die in die Vergangenheit reist, kann abgefangen (gehandlet) werden.

Ausnahmen Werfen

```
raise : exn -> 'a
```

- `raise`
  - ▷ kann an einer beliebigen Stelle in einem Ausdruck E vorkommen
  - ▷ liefert nichts zurück
  - ▷ simuliert nur einen Rückgabewert für den Wert von E
    $\Longrightarrow$ der virtuelle Rückgabewert hat den Typ von E
    $\Longrightarrow$ der Rückgabetyp von `raise` muss polymorphisch sein

```
1 + (raise Div);
uncaught exception divide by zero raised at: stdIn:363.12-363.15
1::(raise Div);
uncaught exception divide by zero raised at: stdIn:263.1-263.4
```

Ausnahmen Behandeln

$$
\begin{array}{rcl}
E \texttt{ handle } P_1 & => & E_1 \\
\mid P_2 & => & E_2 \\
\ldots & => & \\
\mid P_n & => & E_n
\end{array}
$$

- P_1, P_2, ..., P_n sind Muster ähnlich wie bei einem `case` Ausdruck.
- Terminiert das Auswerten von E normal, wird dessen Wert geliefert
- Wirft das Auswerten von E eine Ausnahme ex, liefert den Ausdruck den Wert von E_i, wenn P_i das erste passende Muster ist.
  $\Longrightarrow$ E_i müssen den selben Typ wie E haben
- Sonst wird `ex` weitergeworfen und evt. bei der Auswertung eines umgebenden Ausdrucks (insbesondere Funktionsaufrufs) behandelt
- Das Laufzeitsystem behandelt ungefangene Ausnahmen.

Ausnahmen Verwenden

Ausnahmen können verwendet werden:

- zur Fehlerbehandlung
- als “lange Sprünge” (*longjumps*)
- als Berechnungsmechanismen

4.11.1 Ausnahmen zur Fehlerbehandlung

```
fun head l = case l of nil => raise Empty | h::_ => h
fun tail l = case l of nil => raise Empty | _::r => r

fun member x l = if x=head l then true
                 else member x (tail l)
                 handle Empty => false



member 2 [1,2,3];
val it = true : bool
member 4 [1,2,3];
val it = false : bool
```

4.11.2 Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                        | h::r => if x=h then true
                                  else member x r
```

Ausnahmen als Longjumps

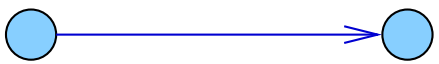
```
fun member x l =  case l of nil => false
                        | h::r => if x=h then true
                                  else member x r
```

`member` x $[a, b, c]$

Ausnahmen als Longjumps

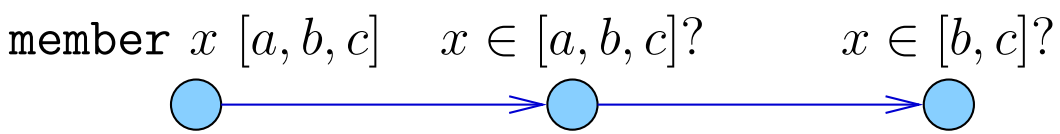
```
fun member x l =  case l of nil => false
                        | h::r => if x=h then true
                                  else member x r
```

member x $[a, b, c]$ $x \in [a, b, c]$?



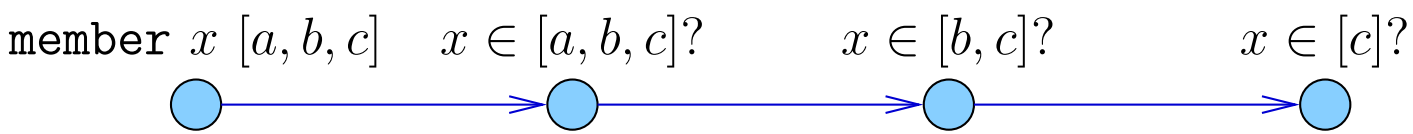
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                        | h::r => if x=h then true
                                  else member x r
```



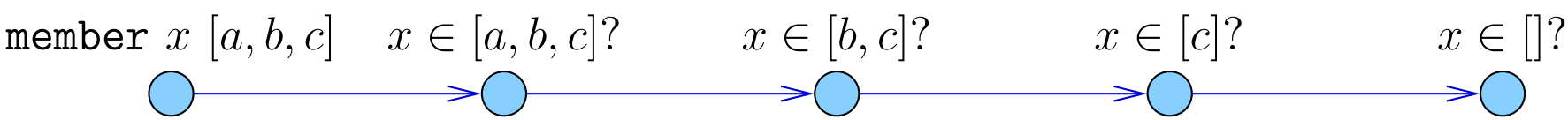
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                        | h::r => if x=h then true
                                  else member x r
```



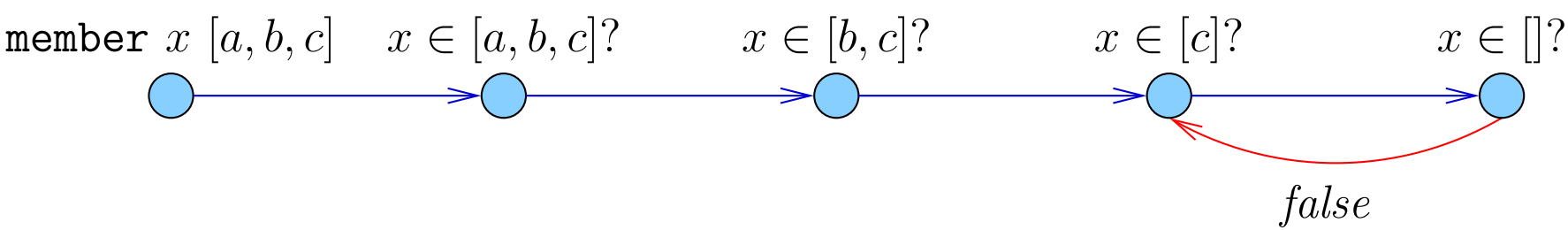
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



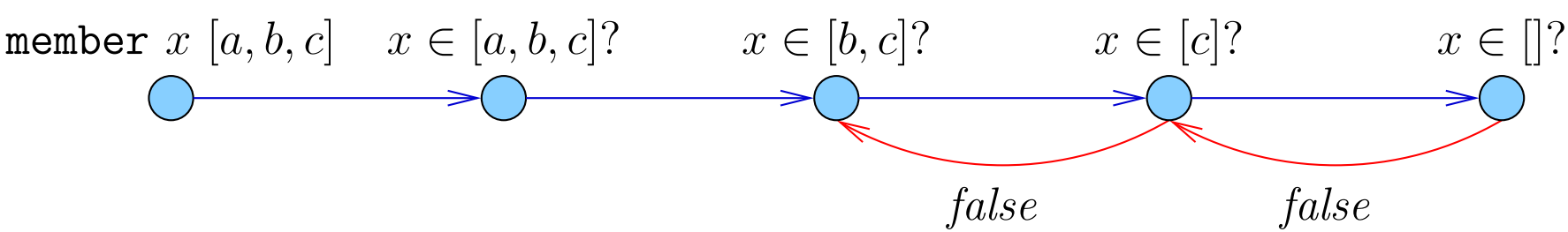
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                        | h::r => if x=h then true
                                  else member x r
```



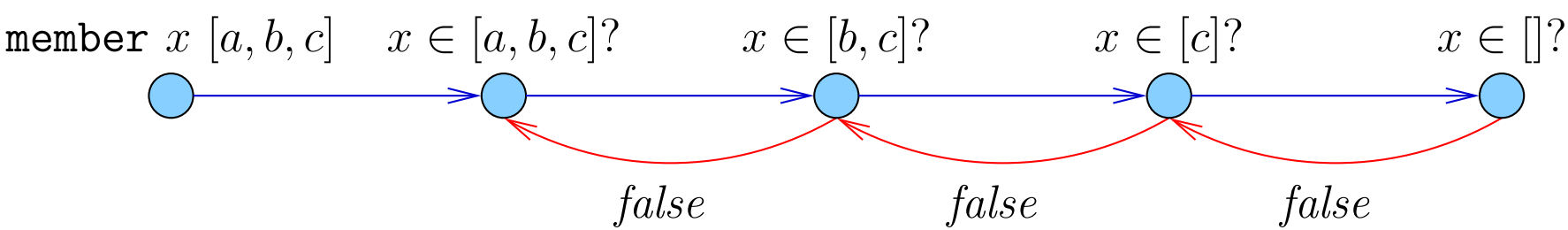
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



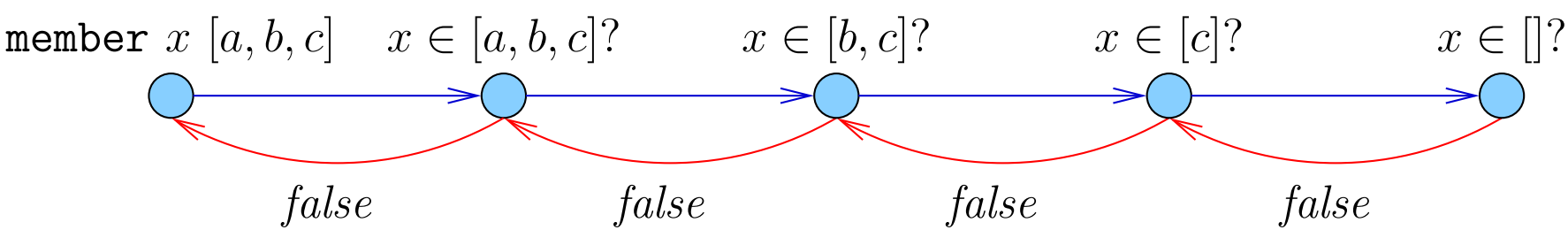
Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



Ausnahmen als Longjumps

```
fun member x l =  case l of nil => false
                          | h::r => if x=h then true
                                    else member x r
```



Ausnahmen als Longjumps

```
exception Result of bool

fun member x list =
  let fun search l = case l of nil => raise (Result false)
                             | h::r => if h=x then
                                         raise (Result true)
                                       else search r
  in search list handle (Result r) => r
  end
```