Fall 3: `x = x1::xs` $\wedge$ `y = y1::ys` $\wedge$ `x1` $\leq$ `y1`.

Wir schließen:

```
sorted (merge x y)   =  sorted (merge (x1::xs) (y1::ys))
                     =  sorted (x1 :: merge xs y)
                     =  ...
```

Fall 3.1: `xs = []`

Wir schließen:

```
                ...  =  sorted (x1 :: merge [] y)
                     =  sorted (x1 :: y)
                     =  sorted y
                     =  true     :-)
```

Fall 3.2: `xs = x2::xs'` $\wedge$ `x2` $\leq$ `y1`.

Insbesondere gilt: `x1` $\leq$ `x2` $\wedge$ `sorted xs`.

Wir schließen:

```
...  =  sorted (x1 :: merge (x2::xs') y)
     =  sorted (x1 :: x2 :: merge xs' y)
     =  sorted (x2 :: merge xs' y)
     =  sorted (merge xs y)
     =  true   nach Induktionsannahme   :-)
```

Fall 3.3: `xs = x2::xs'` $\wedge$ `x2` $>$ `y1`.

Insbesondere gilt: `x1` $\leq$ `y1` $<$ `x2` $\wedge$ `sorted xs`.

Wir schließen:

```
...  =  sorted (x1 :: merge (x2::xs') (y1::ys))
     =  sorted (x1 :: y1 :: merge xs ys)
     =  sorted (y1 :: merge xs ys)
     =  sorted (merge xs y)
     =  true   nach Induktionsannahme   :-)
```

Fall 4: `x = x1::xs` $\wedge$ `y = y1::ys` $\wedge$ `x1` $>$ `y1`.

Wir schließen:

```
sorted (merge x y)   =   sorted (merge (x1::xs) (y1::ys))
                     =   sorted (y1 :: merge x ys)
                     =   ...
```

Fall 4.1: `ys = []`

Wir schließen:

```
                ...  =   sorted (y1 :: merge x [])
                     =   sorted (y1 :: x)
                     =   sorted x
                     =   true     :-)
```

Fall 4.2: `ys = y2::ys'` $\wedge$ `x1` $>$ `y2`.

Insbesondere gilt: `y1` $\leq$ `y2` $\wedge$ `sorted ys`.

Wir schließen:

```
...  =  sorted (y1 :: merge x (y2::ys'))
     =  sorted (y1 :: y2 :: merge x ys')
     =  sorted (y2 :: merge x ys')
     =  sorted (merge x ys)
     =  true   nach Induktionsannahme   :-)
```

Fall 4.3: `ys = y2::ys'` $\wedge$ `x1` $\leq$ `y2`.

Insbesondere gilt: `y1` $<$ `x1` $\leq$ `y2` $\wedge$ `sorted ys`.

Wir schließen:

```
...  =  sorted (y1 :: merge (x1::xs) (y2::ys'))
     =  sorted (y1 :: x1 :: merge xs ys)
     =  sorted (x1 :: merge xs ys)
     =  sorted (merge x ys)
     =  true   nach Induktionsannahme   :-))
```

Diskussion:

- Wieder steht der Beweis unter dem Vorbehalt, dass alle Aufrufe der Funktionen `sorted` und `merge` terminieren :-)
- Als zusätzliche Technik benötigten wir Fallunterscheidungen über die verschiedenen Möglichkeiten für Argumente in den Aufrufen :-)
- Die Fallunterscheidungen machten den Beweis länglich :-(

  // Der Fall $n = 0$ ist tatsächlich überflüssig,

  // da er in den Fällen 1 und 2 enthalten ist :-)

Ausblick:

- In Programmoptimierungen möchten wir gelegentlich Funktionen austauschen, z.B.

  ```
  comp (map f) (map g) = map (comp f g)
  ```

- Offenbar stehen rechts und links des Gleichheitszeichens Funktionen, deren Gleichheit Ocaml nicht überprüfen kann

  $\Longrightarrow$

  Die Logik benötigt einen stärkeren Gleichheitsbegriff :-)

- Wir erweitern die Ocaml-Gleichheit `=` so dass gilt:

  - `x = y` falls `x` und `y` nicht terminieren ;-)
  - `f = g` falls `f x = g x` für alle `x`

    $\Longrightarrow$ extensionale Gleichheit

Wir haben:

- Seien der Typ von e_1, e_2 funktionsfrei. Dann gilt:

$$\frac{e_1 = e_2 \qquad\qquad e_1 \quad \text{terminiert}}{e_1 = e_2 \;\Rightarrow\; \texttt{true}}$$

- Außerdem gibt es eine stark vereinfachte Substitutionsregel:

$$\frac{e_1 = e_2}{e[e_1/x] = e[e_2/x]}$$

// Die Annahmen über Terminierung entfallen :-)

$\Longrightarrow$ Die Beweistechnik vereinfacht sich :-))

7 Das Modulsystem von OCAML

- → Strukturen
- → Signaturen
- → Information Hiding
- → Funktoren
- → Getrennte Übersetzung

7.1 Module oder Strukturen

Zur Strukturierung großer Programmsysteme bietet Ocaml Module oder Strukturen an:

```
module Pairs =
  struct
    type 'a pair = 'a * 'a
    let pair (a,b) = (a,b)
    let first (a,b) = a
    let second (a,b) = b
  end
```

Auf diese Eingabe antwortet der Compiler mit dem Typ der Struktur, einer Signatur:

```
module Pairs :
  sig
    type 'a pair = 'a * 'a
    val pair : 'a * 'b -> 'a * 'b
    val first : 'a * 'b -> 'a
    val second : 'a * 'b -> 'b
  end
```

Die Definitionen innerhalb der Struktur sind außerhalb nicht sichtbar:

```
# first;;
Unbound value first
```

Zugriff auf Komponenten einer Struktur:

Über den Namen greift man auf die Komponenten einer Struktur zu:

```
# Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

So kann man z.B. mehrere Funktionen gleichen Namens definieren:

```
# module Triples = struct
    type 'a triple = Triple of 'a * 'a * 'a
    let first (Triple (a,_,_)) = a
    let second (Triple (_,b,_)) = b
    let third (Triple (_,_,c)) = c
  end;;
...
```

```
...
module Triples :
sig
  type 'a triple = Triple of 'a * 'a * 'a
  val first : 'a triple -> 'a
  val second : 'a triple -> 'a
  val third : 'a triple -> 'a
end
# Triples.first;;
- : 'a Triples.triple -> 'a = <fun>
```

... oder mehrere Implementierungen der gleichen Funktion:

```
# module Pairs2 =
  struct
    type 'a pair = bool -> 'a
    let pair (a,b) = fun x -> if x then a else b
    let first ab = ab true
    let second ab = ab false
  end;;
```

Öffnen von Strukturen

Um nicht immer den Strukturnamen verwenden zu müssen, kann man alle Definitionen einer Struktur auf einmal sichtbar machen:

```
# open Pairs2;;
# pair;;
- : 'a * 'a -> bool -> 'a = <fun>
# pair (4,3) true;;
- : int = 4
```

Sollen die Definitionen des anderen Moduls Bestandteil des gegenwärtigen Moduls sein, dann macht man sie mit `include` verfügbar ...

```
# module A = struct let x = 1 end;;
module A : sig val x : int end
# module B = struct
    open A
    let y = 2
  end;;
module B : sig val y : int end
# module C = struct
    include A
    include B
  end;;
module C : sig val x : int val y : int end
```

Geschachtelte Strukturen

Strukturen können selbst wieder Strukturen enthalten:

```
module Quads  = struct
    module Pairs = struct
        type 'a pair = 'a * 'a
        let pair (a,b) = (a,b)
        let first (a,_) = a
        let second (_,b) = b
      end
    type 'a quad = 'a Pairs.pair Pairs.pair
    let quad (a,b,c,d) =
        Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
    ...
```

```
    ...
    let first q = Pairs.first (Pairs.first q)
    let second q = Pairs.second (Pairs.first q)
    let third q = Pairs.first (Pairs.second q)
    let fourth q = Pairs.second (Pairs.second q)
  end

# Quads.quad (1,2,3,4);;
- : (int * int) * (int * int) = ((1,2),(3,4))
# Quads.Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

7.2 Modul-Typen oder Signaturen

Mithilfe von Signaturen kann man einschränken, was eine Struktur nach außen exportiert.

Beispiel:

```
module Count =
  struct
    let count = ref 0
    let setCounter n  = count := n
    let getCounter () = !count
    let incCounter () = count := !count+1
  end
```

Der Zähler ist nach außen sichtbar, zugreifbar und veränderbar:

```
# !Count.count;;
- : int = 0
# Count.count := 42;;
- : unit = ()
```

Will man, daß nur die Funktionen der Struktur auf ihn zugegreifen können, benutzt man einen Modul-Typ oder Signatur:

```
module type Count =
  sig
    val setCounter : int -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> int
  end
```

Die Signatur enthält den Zähler selbst nicht :-)

Mit dieser Signatur wird die Schnittstelle der Struktur eingeschränkt:

```
# module SafeCount : Count = Count;;
module SafeCount : Count
# SafeCount.count;;
Unbound value Safecount.count
```

Die Signatur bestimmt, welche Definitionen exportiert werden. Das geht auch direkt bei der Definition:

```
module Count1 : Count =
  struct
    val count = ref 0
    fun setCounter n  = count := n
    fun getCounter () = !count
    fun incCounter () = count := !count+1
  end

# !Count1.count;;
Unbound value Count1.count
```

Signaturen und Typen

Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein :-)

Dadurch werden deren Typen spezialisiert:

```
module type A1 = sig
    val f : 'a -> 'b -> 'b
  end
module type A2 = sig
    val f : int -> char -> int
  end
module A = struct
    let f x y = x
  end
```

```
# module A1 : A1 = A;;
Signature mismatch:
Modules do not match: sig val f : 'a -> 'b -> 'a end
                                  is not included in A1
Values do not match:
  val f : 'a -> 'b -> 'a
is not included in
  val f : 'a -> 'b -> 'b
# module A2 : A2 = A;;
module A2 : A2
# A2.f;;
- : int -> char -> int = <fun>
```

7.3 Information Hiding

Aus Gründen der Modularität möchte man oft verhindern, dass die Struktur exportierter Typen einer Struktur von außen sichtbar ist.

Beispiel:

```
module ListQueue = struct
  type 'a queue = 'a list
  let empty_queue () = []
  let is_empty = function
      [] -> true | _  -> false
  let enqueue xs y = xs @ [y]
  let dequeue (x::xs) = (x,xs)
end
```

Mit einer Signatur kann man die Implementierung einer Queue verstecken:

```
module type Queue = sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end
```

```
# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
  'b queue = 'b Queue.queue
```

$\Longrightarrow$

Das Einschränken per Signatur genügt, um die wahre Natur des Typs `queue` zu verschleiern :-)

Soll der Datentyp mit seinen Konstruktoren dagegen exportiert werden, wiederholen wir seine Definition in der Signatur:

```
module type Queue =
sig
   type 'a queue = Queue of ('a list * 'a list)
   val empty_queue : unit -> 'a queue
   val is_empty : 'a queue -> bool
   val enqueue : 'a -> 'a queue -> 'a queue
   val dequeue : 'a queue -> 'a option * 'a queue
end
```

7.4 Funktoren

Da in Ocaml fast alles höherer Ordnung ist, wundert es nicht, dass es auch Strukturen höherer Ordung gibt: die Funktoren.

- Ein Funktor bekommt als Parameter eine Folge von Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Argumente des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.

Wir legen zunächst per Signatur die Eingabe und Ausgabe des Funktors fest:

```
  module type Enum = sig
    type enum
    val null : enum
    val incr : enum -> enum
    val int_of_enum : enum -> int
  end
  module type Counter = sig
    type counter
    val incCounter : unit -> unit
    val getCounter : unit -> counter
    val int_of_counter : counter -> int
  end
...
```

```
...
# module GenCounter (Enum : Enum) : Counter =
  struct
    open Enum
    type counter = enum
    let count = ref null
    let incCounter() = count := incr(!count)
    let getCounter() = !count
    let int_of_counter x = int_of_enum x
  end;;
module GenCounter : functor (Enum : Enum) -> Counter
```

Jetzt können wir den Funktor auf eine Struktur anwenden und erhalten eine neue Struktur ...

```
  module PlusEnum = struct
    type enum = int
    let null = 0
    let incr x = x+1
    let int_of_enum x = x
  end

# module PlusCounter = GenCounter (PlusEnum);;
module PlusCounter : Counter

# PlusCounter.incCounter ();
  PlusCounter.incCounter ();
  PlusCounter.int_of_counter (PlusCounter.getCounter ());;
- : int = 2
```