Formal Verification of OCaml Programs

Funktionale Programmierung (Technische Universität München)

Formal Verification of OCaml Programs

- allow us to make sure an OCaml program behaves as it should
- MiniOCaml: allowing us to only consider a fragment of OCaml which only includes base types int, bool, tuples, lists, and recursive functions only at top level
- **OCaml Grammar**
  - expressions E :: = const (e.g. 5) | name (e.g. x or f) | op_1 *E* (e.g. -e) | *E*_1 op_2 *E*_2 (e.g. e_1 + e_2) | (*E*_1,...,*E_k*) (e.g.) | let name = *E*_1 in *E*_0 (e.g. let construct with name equals e_1 used in e_0) | match *E* with *P*_1 -> *E*_1 | ... | *P_k* -> *E_k* | fun name -> *E* | *E E*_1 (e.g. function application is a binary operation between function e and argument e_1)
  - pattern P :: = const | name | (P_1, ..., P_k) | P_1 :: P_2 (P_1 = head, P_2 = tail)
  - well typed: pair of int * list, e.g. (1, [true; false]) ; (1 [true; false]) is not well typed
  - every name may occur once
  - if-conditions not included as representable by `match ... with true -> ... | false -> ...`
  - program consists of a set of mutually (i.e. functional declarations can be used in any other function) recursive global definitions:

```
let rec f1 = E1
    and f2 = E2
    ...
    and fm = Em
```

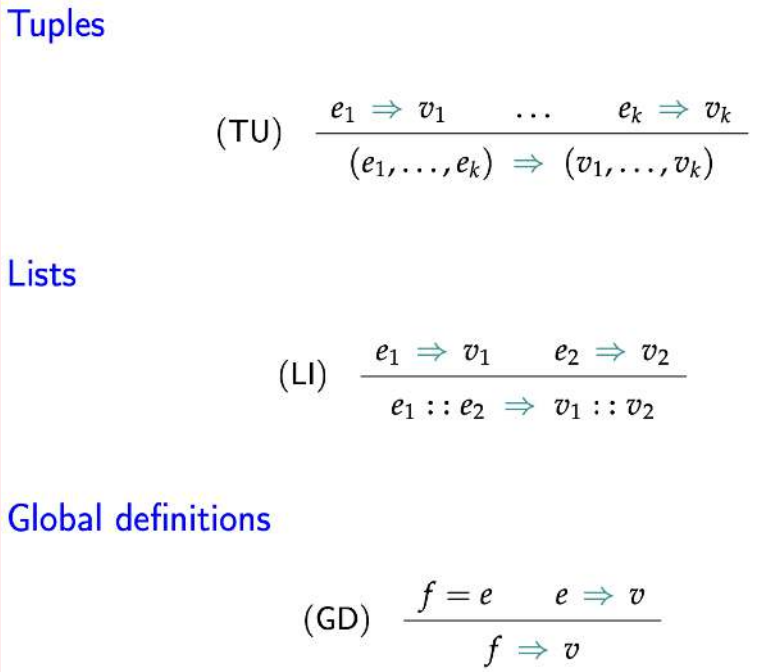
  - **a value is an expression that cannot be further evaluated**, values are V :: = const | fun name_1 ... name_k → *E* | (V_1, ..., V_k) | [] | V_1 :: V_2
- **Big-Step Proofs** operational semantics defines a relation $e \Rightarrow v$ between expressions and their values, i.e. during execution OCaml takes an expression *e*

and evaluates it (computes all steps) until no further computation can be performed, thus OCaml reached a value where $v \Rightarrow v$ holds for every value v

- Hints: When evaluating an expression such as `bar foo 10 32 ==> 34` we evaluate functions from left to right due to their definition, thus we first evaluate bar with input foo and 10 and their result will be applied to 32

```
let rec check l = match l with [] -> true | 0::t | 1::t -> check t
    | _ -> false
let rec fl f a l = match l with [] -> a | h::t -> fl f (f a h) t
let foo a b = 2 * b :: a
let bar f x = match f [] 1 with [] -> (fun a -> a)
    | h::_ -> (fun a -> a + h)
let g a b = a && (b = 0 || b = 1)
```

- Hints: pi_declarations allow us to reuse big-step proof trees and tau_declarations allow us to reuse simple expressions
- **Rules :** goal is to arrive at expression displayed in denominator → **SPICK**



Local definitions

$$\text{(LD)}\quad \frac{e_1 \Rightarrow v_1 \qquad e_0[v_1/x] \Rightarrow v_0}{\texttt{let } x = e_1 \texttt{ in } e_0 \Rightarrow v_0}$$

Function calls

$$\text{(APP)}\quad \frac{e \Rightarrow \texttt{fun } x \texttt{ -> } e_0 \qquad e_1 \Rightarrow v_1 \qquad e_0[v_1/x] \Rightarrow v_0}{e\ e_1 \Rightarrow v_0}$$

$$\text{(APP)}\quad \frac{e_0 \Rightarrow \texttt{fun } x_1 \ldots x_k \texttt{ -> } e \qquad e_1 \Rightarrow v_1 \ldots e_k \Rightarrow v_k \qquad e[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v}{e_0\ e_1 \ldots e_k \Rightarrow v}$$

repeated application of function calls result in a rule for functions with multiple arguments

- difference between global and local definitions: local definitions are definitions that are only valid for the expression for which they are defined for, e.g. `let x = e_1 in e_0` the local definition *x* can only be used in *e_0* ; therefore proofing can be interpreted as evaluating the expression *e_1* of *x* into *v_1* and then applying the evaluation *v_1* to *e_0* for every *x* in *e_0* by replacing *x* with the evaluation of *v_1* resulting in *v_0*
- function calls: expression e applied to *e_1*, OCaml will first figure out function of e through `fun x → e_0` , then OCaml evaluates the value *v_1* of the given argument expression *e_1* , then we again apply the value *v_1* to the expression *e_0* by replacing *v_1* for x to obtain *v_0*

Pattern Matching

$$\text{(PM)}\quad \frac{e_0 \Rightarrow v' \equiv p_i[v_1/x_1, \ldots, v_k/x_k] \qquad e_i[v_1/x_1, \ldots, v_k/x_k] \Rightarrow v}{\texttt{match } e_0 \texttt{ with } p_1 \texttt{ -> } e_1 \texttt{ | } \ldots \texttt{ | } p_m \texttt{ -> } e_m \Rightarrow v}$$

— given that v' does not match any of the patterns $p_1, \ldots, p_{i-1}$

;-)

- first OCaml will evaluate *e_0* to *v'*, then *v'* will be compared to the patterns to choose the appropriate expression *e_i* matching the *v'*, matching means the values *x_i* can be replaced by the values of *v' (v_1 ... v_k)* such that they

match, the corresponding *e_i* will then be evaluated to *v* by replacing its *x_i*'s with the corresponding values *v_i*

Built-in operators

$$(\text{OP}) \quad \frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2 \qquad v_1 \,\text{op}\, v_2 \Rightarrow v}{e_1 \,\text{op}\, e_2 \Rightarrow v}$$

Unary operators are treated analogously.

- unary operations have only one operand, i.e. a single input, e.g. function f: A → A; likewise unary functions are functions with only one argument
- *e_1* op *e_2* could be 5 + 6 to evaluated to value *v* ⇒ 11
- *v_1 op v_2* ⇒ *v* is given by the "oracle" which tells me its evaluation

- **Examples**

The built-in equality operator

$$v = v \Rightarrow \texttt{true}$$

$$v_1 = v_2 \Rightarrow \texttt{false}$$

given that v, v_1, v_2 are values that do not contain functions, and v_1, v_2 are syntactically different.

Example 1

$$\dfrac{\dfrac{\texttt{17} \Rightarrow \texttt{17} \qquad \texttt{4} \Rightarrow \texttt{4} \qquad \texttt{17+4} \Rightarrow \texttt{21}}{\texttt{17+4} \Rightarrow \texttt{21}} \qquad \texttt{21} \Rightarrow \texttt{21} \qquad \texttt{21=21} \Rightarrow \texttt{true}}{\texttt{17 + 4 = 21} \Rightarrow \texttt{true}}$$

the exact same functions are still not considered equal by OCaml

- if axioms *v* ⇒ *v* are omitted

$$\dfrac{\dfrac{17+4 \Rightarrow 21}{17+4 \Rightarrow 21} \qquad 21=21 \Rightarrow \text{true}}{17 + 4 = 21 \Rightarrow \text{true}}$$

Example 2

```
let rec f = fun x -> x+1
    and s = fun y -> y*y
```

f increments and s squares

$$\dfrac{\dfrac{f = \text{fun } x \text{ -> } x+1 \qquad 16+1 \Rightarrow 17}{f \Rightarrow \text{fun } x \text{ -> } x+1 \qquad 16+1 \Rightarrow 17} \qquad \dfrac{s = \text{fun } y \text{ -> } y*y \qquad 2*2 \Rightarrow 4}{s \Rightarrow \text{fun } y \text{ -> } y*y \qquad 2*2 \Rightarrow 4} \qquad 17+4 \Rightarrow 21}{f\ 16 + s\ 2 \Rightarrow 21}$$

// uses of $v \Rightarrow v$ have mostly been omitted

- in case the evaluation is not yet known, as it might be too complex, then one can simply continue to evaluate expressions until expressions can no longer be evaluated

Example 3

```
let rec app = fun x y -> match x
        with [] -> y
         | h::t -> h :: app t y
```

Claim: `app (1::[]) (2::[]) ⇒ 1::2::[]`

Proof

$$\dfrac{\dfrac{\texttt{app = fun x y -> ...}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}} \quad \dfrac{\texttt{[]} \Rightarrow \texttt{[]} \quad \texttt{2::[]} \Rightarrow \texttt{2::[]}}{\texttt{match [] ...} \Rightarrow \texttt{2::[]}}}{\dfrac{\texttt{app [] (2::[])} \Rightarrow \texttt{2::[]}}{\dfrac{\texttt{app = fun x y -> ...} \quad \texttt{1 :: app [] (2::[])} \Rightarrow \texttt{1::2::[]}}{\dfrac{\texttt{app} \Rightarrow \texttt{fun x y -> ...} \quad \texttt{match 1::[] ...} \Rightarrow \texttt{1::2::[]}}{\texttt{app (1::[]) (2::[])} \Rightarrow \texttt{1::2::[]}}}}}$$

// uses of $v \Rightarrow v$ have mostly been omitted

- **Proving program terminates**
  - in order to prove that the evaluation of a function terminates for a particular argument values, it suffices to prove that there are values to which corresponding function calls can be evaluated
  - only plausible way of proving such claims are inductions

Example Claim

app l_1 l_2 terminates for all list values l_1, l_2.

Proof

Induction on the length n of the list l_1.

$n = 0$ I.e., l_1 = []. Then

$$\dfrac{\dfrac{\texttt{app = fun x y -> } \cdots}{\texttt{app} \Rightarrow \texttt{fun x y -> } \cdots} \quad \texttt{match [] with [] -> } l_2 \texttt{ | ...} \Rightarrow l_2}{\texttt{app [] } l_2 \Rightarrow l_2}$$

based on definition of function it seems plausible to perform induction on first argument list *l_1* since the *app* function only looks at *l_1*

$n > 0$: I.e., l_1 = `h::t`.

In particular, we assume that the claim already holds for all shorter lists. Then we have:

$$\texttt{app t } l_2 \Rightarrow l$$

for some l. We deduce

$$\dfrac{\dfrac{\texttt{app = fun x y -> ...}}{\texttt{app} \Rightarrow \texttt{fun x y -> ...}} \quad \dfrac{\dfrac{\texttt{app t } l_2 \Rightarrow l}{\texttt{h :: app t } l_2 \Rightarrow \texttt{h :: } l}}{\texttt{match h::t with } \cdots \Rightarrow \texttt{h :: } l}}{\texttt{app (h::t) } l_2 \Rightarrow \texttt{h :: } l}$$

based on induction hypothesis we know that if the first and second list have at least one element the function *app* must return a list that contains the elements of the 1st and 2nd list

- big-step semantics verify correctness of optimization transformations, prove correctness of assertions about functional programs, suggest to consider expressions as specifications of values
- values can only be equal if they don't contain functions

  c :: = const | (*C*1,...,*Ck*) | [] | *C*1::*C*2 , thus only of the type *c* ::= bool | int | unit | *c*1∗...∗*ck* | *c*list

Extension of Equality

The equality = of Ocaml is extended to expression which may not terminate, and functions.

Non-termination

$$\frac{e_1, e_2 \qquad \text{both not terminating}}{e_1 = e_2}$$

Termination

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2 \qquad v_1 = v_2}{e_1 = e_2}$$

in order to prove two expressions are the same we will show that both terminate and evaluate into the same value, thus we evaluate them until they can no longer be evaluated, i.e. correspond to a value, and compare those two values

- if two expressions don't terminate we consider them equal as it wouldn't matter which one we apply in order to have a non-terminating expression

Structured values

$$\frac{v_1 = v_1' \ \ldots \ v_k = v_k'}{(v_1, \ldots, v_k) = (v_1', \ldots, v_k')}$$

$$\frac{v_1 = v_1' \qquad v_2 = v_2'}{v_1 :: v_2 = v_1' :: v_2'}$$

tuples are equivalent if each of its components are equivalent

Functions

$$\frac{e_1[v/x_1] = e_2[v/x_2] \quad \text{for all} \quad v}{\texttt{fun } x_1 \texttt{ -> } e_1 = \texttt{fun } x_2 \texttt{ -> } e_2}$$

$\Longrightarrow$ extensional equality

functions are considered equivalent if they evaluate to the same for all values as arguments

We have:

$$\frac{e \Rightarrow v}{e = v}$$

Assume that the type of e_1, e_2 is functionfree. Then

$$\frac{e_1 = e_2 \qquad e_1 \text{ terminates}}{e_1 = e_2 \Rightarrow \texttt{true}}$$

$$\frac{e_1 = e_2 \Rightarrow \texttt{true}}{e_1 = e_2 \qquad e_i \text{ terminates}}$$

- key idea of proving the equality of two things is the following lemma

Substitution Lemma

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

We deduce for functionfree expressions e:

$$\frac{e_1 = e_2 \qquad e[e_1/x] \text{ terminate}}{e[e_1/x] = e[e_2/x] \Rightarrow \texttt{true}}$$

if there exist two expressions *e_1* and *e_2* that have been proven to be equal, then replacing the *x* of any expression *e* with *e_1* and *e_2* will also result in equal expressions; this allow us to deduce that these expressions terminate if *e_1* is equal to *e_2* and both terminate

- lemma tells us in every context, all occurrences of *e_1* are replaceable by *e_2* whenever *e_1* and *e_2* represent the same value
- Equality of expressions, useful for proofs

Simplification of local definitions

$$\frac{e_1 \text{ terminates}}{\texttt{let } x = e_1 \texttt{ in } e \;=\; e[e_1/x]}$$

Simplification of function calls

$$\frac{e_0 \;=\; \texttt{fun } x \texttt{->} e \qquad e_1 \text{ terminates}}{e_0\ e_1 \;=\; e[e_1/x]}$$

repeatedly applying function calls results in rule for functions with multiple arguments

$$\frac{e_0 \;=\; \texttt{fun } x_1 \ldots x_k \texttt{->} e \qquad e_1, \ldots, e_k \text{ terminate}}{e_0\ e_1 \ldots e_k \;=\; e[e_1/x_1, \ldots, e_k/x_k]}$$

- note expressions are meant to represent values

- Proving `let` rule

Since e_1 terminates, there is a value v_1 with

$$e_1 \Rightarrow v_1$$

Due to the Substitution Lemma, we have:

$$e[v_1/x] = e[e_1/x]$$

Case 1: $e[v_1/x]$ terminates.

Then a value v exists with

$$e[v_1/x] \Rightarrow v$$

since *e[v_1/x]* terminates it must have a value as it can be evaluated until it can't be further evaluated, thus resulting in a value, we can deduce *e[v_1/x]* $\Rightarrow$ *v*

Then

$$e[e_1/x] = e[v_1/x] = v$$

Because of the big-step semantics, however, we have:

$$\texttt{let } x = e_1 \texttt{ in } e \Rightarrow v \quad \text{and therefore,}$$
$$\texttt{let } x = e_1 \texttt{ in } e = e[e_1/x]$$

Case 2: $e[v_1/x]$ does not terminate.

Then $e[e_1/x]$ does not terminate and neither does $\texttt{let } x = e_1 \texttt{ in } e$.

Accordingly,

$$\texttt{let } x = e_1 \texttt{ in } e = e[e_1/x]$$

since *e[v_1/x]* doesn't terminate, but is equal to *e[e_1/x]* due to the Substitution lemma, we know that *e[e_1/x]* also doesn't terminate, hence doesn't `let x = e_1 in e` which shows that all expressions don't terminate and are considered equal

Rule for pattern matching

match case of expression being the empty list, note that if e_1 doesn't terminate then whole match expression also doesn't terminate

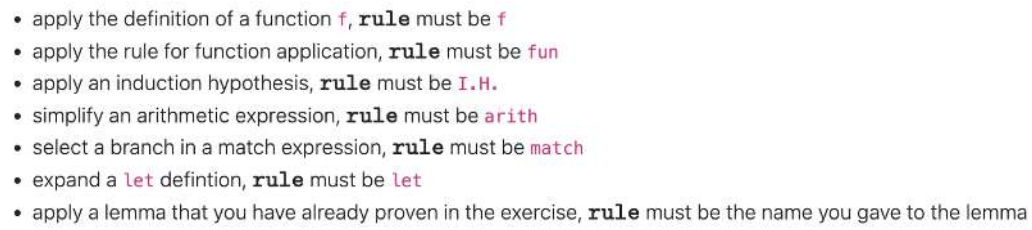
$$\frac{e_0 = []}{\texttt{match } e_0 \texttt{ with } [] \texttt{->} e_1 \texttt{ | } \ldots \texttt{ | } p_m \texttt{->} e_m = e_1}$$

e_0 is not empty, has head and tail, e_0 will be evaluated to a value (list) and therefore can be assumed to terminate, if it wouldn't terminate the equality of the match and expression e_2 would not hold

$$\frac{e_0 \text{ terminates} \qquad e_0 = e_1' :: e_2'}{\texttt{match } e_0 \texttt{ with } [] \texttt{->} e_1 \texttt{ | } x :: xs \texttt{->} e_2 = e_2[e_1'/x, e_2'/xs]}$$

- **Proving OCaml programs / Equational Equivalences/Reasoning**
  - General approach: prove a given statement (= OCaml program) for all possible inputs; do so by proving that the given statement holds for the base input (usually the empty list) and then show that the statement also holds for a list that consists of one more element (usually h :: l)
  - Strategy to prove a statement by induction: setup base case and induction case using induction hypothesis, for each of those cases start with left-side and transform expressions according to their definition until point with no further possible transformations, then continue with rightside of equality statement to be proven and transform it going upwards towards left-side

- strategy: simply keep smaller functions, do not evaluate them to simplify things and continue to work with it and see where you get, e.g. proving equality of `(HL) fl (+) acc1 (map foo l) = acc1 + sum (fun x a -> a + foo x) l`
- also note that `(fun)` and `(f functionname)` can be done in one step
- To provide description on each transformation step



- In case one of the functions conatins accumulator → Funktionen auf Akkumulator prüfen
  - Wir müssen das Akkumulatorargument verallgemeinern, wenn es sich in Rekursionsaufrufen verändert ; meistens gilt dies auch für Expressions bei denen Konstanten verwendet werden
  - Funktion mit Akkumulator, Akkumulator ausschreiben
    - Funktion ohne Akkumulator zu Beginn mit acc * oder acc + annotieren

1. Beispiel

```
mul c (sum l 0) 0 = c * summa l
```

Funktion mal und sum haben einen acc, die Funktion summa l nicht

```
mul c (sum l acc1) acc2 = acc2 + c * (acc1 + summa l)
```

Linke Seite schreibe den acc also null mit acc1 aus, Rechte Seite acc + zu Beginn

2. Beispiel

```
fold_left (+) 0 l = fold_right (+) l 0
```

Funktion fold_left hat einen acc, die Funktion fold_right nicht

```
fold_left (+) a l = a + fold_right (+) l 0
```

Wenn das Argument nicht mit 0, sondern beliebigen Wert a initialisiert wird, wird nicht mehr sum(l) sondern, sum(l) + a berechnet → Deshalb Linke Seite a ausschreiben, Rechte Seite acc +

3. Beispiel

```
fact_aux 1 n = fact n
```

Linke Seite hat acc in Form von x=1, Gleichheit kann nur für x=1 bewiesen werden. X verändert sich aber im Laufe der Aufrufe und fact_aux wird auf x multipliziert (x *n!)

```
fact_aux acc n = acc * fact n
```

Linke Seite acc ausschreiben, Rechte Seite acc* zu Beginn schreiben

- Example Proofs

Example 1

```
let rec app = fun x -> fun y -> match x
                  with [] -> y
                  | h::t -> h :: app t y
```

We want to verify that

(1) `app x [] = x` for all lists x.

(2) `app x (app y z) = app (app x y) z` for all lists x, y, z.

Note that statement (2) is associativity.

Idea: Induction on the length n of `x`

$n = 0$ Then `x = []` holds.

We deduce:

```
app x []  =  app [] []
          =  match [] with [] -> [] | h::t -> h :: app t []
          =  []
          =  x
```

$n > 0$ Then: `x = h::t` where `t` has length $n - 1$.

We deduce:

```
app x []  =  app (h::t) []
          =  match h::t with [] -> [] | h::t -> h :: app t []
          =  h :: app t []
          =  h :: t      by induction hypothesis
          =  x
```

Analogously we proceed for assertion (2) ...

$n = 0$ Then: `x = []`

We deduce:

```
app x (app y z)  =  app [] (app y z)
                 =  match [] with [] -> app y z | h::t -> ...
                 =  app y z
                 =  app (match [] with [] -> y | ...) z
                 =  app (app [] y) z
                 =  app (app x y) z
```

$\boxed{n > 0}$ Then `x = h::t` where `t` has length $n - 1$.

We deduce:

```
app x (app y z)  =  app (h::t) (app y z)
                 =  match h::t with [] -> app y z
                        | h::t -> h :: app t (app y z)
                 =  h :: app t (app y z)
                 =  h :: app (app t y) z   by induction hypothesis
                 =  app (h :: app t y) z
                 =  app (match h::t with [] -> []
                               | h::t -> h :: app t y) z
                 =  app (app (h::t) y) z
                 =  app (app x y) z
```

- note that for our induction proofs all occurring function calls have to terminate; it suffices to prove that for all `x, y` there exists some `v` such that `app x y => v` which we already proved by induction
- Detour to Induction Proofs:
  - general idea: we want to proof that some statement holds true for all possible values of a variable of that statement
  - proof strategy: one can prove such a thing by proving it through induction applying the following approach:
    1. Show that the statement holds true for a base case, usually first element of the set of allowed elements, e.g. 0, empty list etc.
    2. In the next step we assume that our statement holds true for some value k and based on this assumption we want to show that if the statement holds true for k then it must also hold true for k+1, i.e. the next element; if we are indeed able to prove this then the statement must hold for all possible values of the given value set because since k can be any value of the given valid value set the statement holds for that one and the next (k+1) and so on...
  - clue: while proving the induction step, i.e. that the statement holds true for k+1, we can use the induction hypothesis which is our

assumption that the given statement is true for k since it must be true if our induction is true

- example:

Example 2

```
let rec rev = fun x -> match x
        with [] -> []
        | h::t -> app (rev t) [h]
let rec rev1 = fun x -> fun y -> match x
        with [] -> y
        | h::t -> rev1 t (h::y)
```

Claim

`rev x = rev1 x []` for all lists `x`.

More generally,

`app (rev x) y = rev1 x y` for all lists `x, y`.

Proof: Induction on the length n of `x`

$n = 0$ Then: `x = []`. We deduce:

```
app (rev x) y  =  app (rev []) y
               =  app (match [] with [] -> [] | ...) y
               =  app [] y
               =  y
               =  match [] with [] -> y | ...
               =  rev1 [] y
               =  rev1 x y
```

as we did in the homework 12: we start with the left-handside `app (rev x) y` and evaluate it as far as possible, here until we obtain `y` and then we further evaluate the right-handside from the bottom up towards the `y`

$n > 0$ Then `x = h::t` where `t` has length $n - 1$.

We deduce (ommitting simple intermediate steps):

```
app (rev x) y  =  app (rev (h::t)) y
               =  app (app (rev t) [h]) y
               =  app (rev t) (app [h] y)    by example 1
               =  app (rev t) (h::y)
               =  rev1 t (h::y)     by induction hypothesis
               =  rev1 (h::t) y
               =  rev1 x y
```

evaluating `rev (h::t))` leads us to the second matching case which returns `app (rev t) [h]` ; interesting part here is that we can use the proof of example 1

- note we again assumed calls of `app`, `rev`, and `rev1` terminate, termination can be proven by induction on length of first arguments

Example 3

```
let rec sorted = fun x -> match x
        with h1::h2::t -> (match h1 <= h2
                     with true -> sorted (h2::t)
                       | false -> false)
           | _          -> true

and merge = fun x -> fun y -> match (x,y)
    with ([],y) -> y
    |    (x,[]) -> x
    | (x1::xs,y1::ys) -> (match x1 <= y1
                  with true -> x1 :: merge xs y
                    | false -> y1 :: merge x ys
```

- proof sorted routine is correct: prove merge function performs correctly, do so we need a predicate indicating whether a list is sorted or not which is handled by recursive function sorted
- sorted takes one list as argument, if empty or consists of only one element (case `_`) it returns true as list necessarily sorted ; else if list contains more than one element case `h1::h2::t` matches (accessing

first, second, and remaining elements), then we check if h1 ≤ h2, if so we check the second and remaining elements, note there is an overlapping check

Claim

conditional statement

`sorted x` $\wedge$ `sorted y` $\rightarrow$ `sorted (merge x y)`

for all lists `x, y`.

Proof: Induction on the sum n of lengthes of `x, y`.

Assume that `sorted x` $\wedge$ `sorted y` holds.

$n = 0$ Then: `x = [] = y`

We deduce:

```
sorted (merge x y)  =  sorted (merge [] [])
                    =  sorted []
                    =  true
```

the claim can help us in our proof, since we can infer from it that `sorted x = true` and `sorted y = true`, we want to show that `sorted (merge x y) = true`

$n > 0$

Case 1: `x = []`.

We deduce:

```
sorted (merge x y)  =  sorted (merge [] y)
                    =  sorted y
                    =  true
```

Case 2: `y = []` analogous.

several cases must be considered: (1 & 2) one of the two lists (x or y) is empty

Case 3: `x = x1::xs ∧ y = y1::ys ∧ x1 ≤ y1.`

We deduce:

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

Case 3.1: `xs = []`

We deduce:

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

further case distinction: `x1 <= x2` which itself results in further case distinctions

Case 3.2: `xs = x2::xs' ∧ x2 ≤ y1.`

In particular: `x1 ≤ x2 ∧ sorted xs.`

We deduce:

```
...  =  sorted (x1 :: merge (x2::xs') y)
     =  sorted (x1 :: x2 :: merge xs' y)
     =  sorted (x2 :: merge xs' y)
     =  sorted (merge xs y)
     =  true   by induction hypothesis
```

- Note that we can infer `x1 <= x2` as we know from our assumption that `x` is sorted; moreover since `x2` is less than every element in `y` since we assume `x2 <= y1` we can rewrite `merge (x2::xs') y` to `x2 :: merge xs' y`
- clue here is applying the induction hypothesis: since `xs` and `y` are in sum shorter lists than `x` and `y` and are also both sorted based on our assumptions merging `xs` and `y` must also return a sorted list, hence `sorted (merge xs y)` must be `true` because a sorted list is obviously sorted → What exactly is the induction hypothesis? according to lecture chat `sorted (merge xs ys) = true`

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

In particular: `x1` $\leq$ `y1` $<$ `x2` $\wedge$ `sorted xs`.

We deduce:

```
...  =  sorted (x1 :: merge (x2::xs') (y1::ys))
     =  sorted (x1 :: y1 :: merge xs ys)
     =  sorted (y1 :: merge xs ys)
     =  sorted (merge xs y)
     =  true   by induction hypothesis
```