Chapter 7
Higher-Order Functions

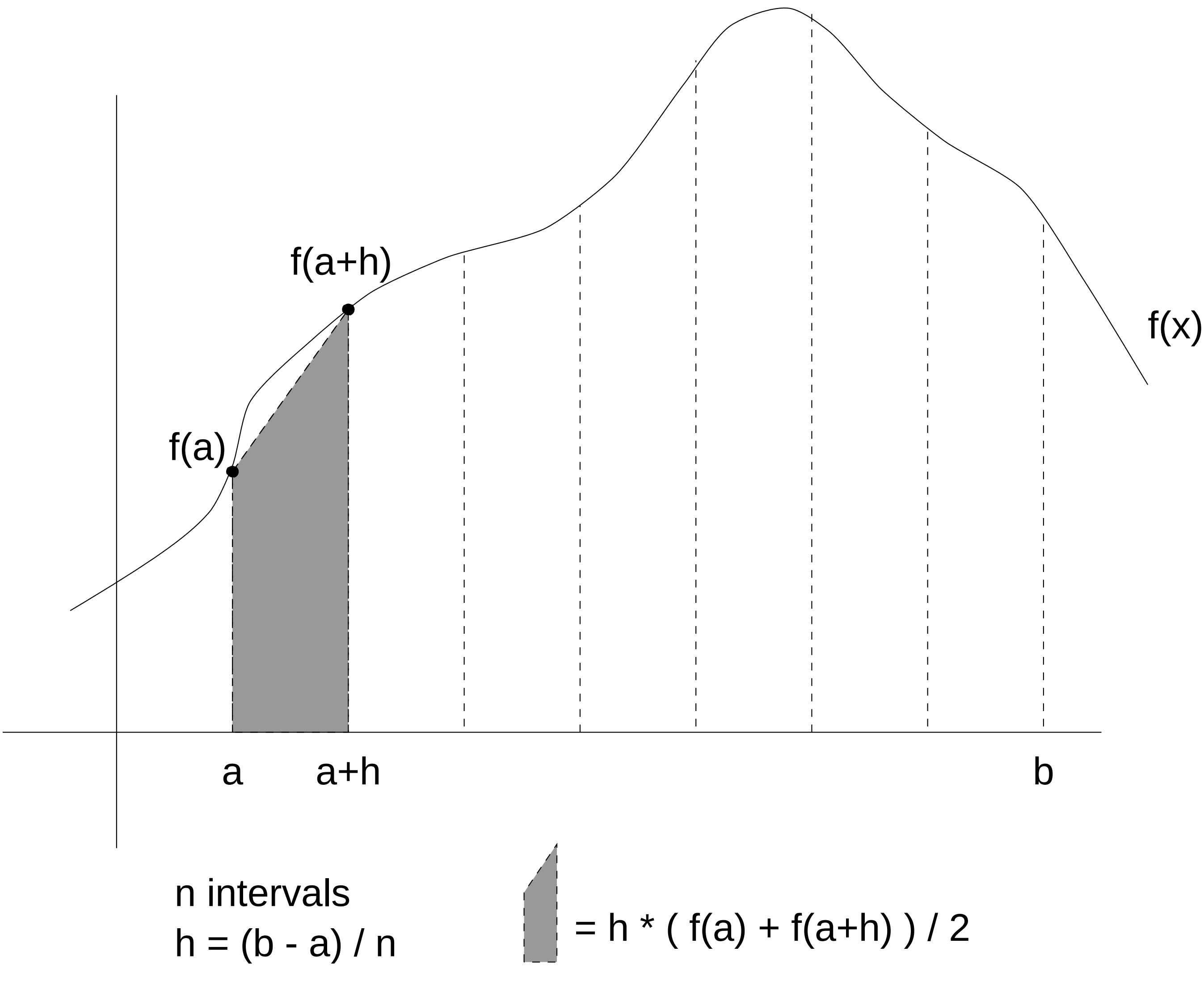
(Version of 27 September 2004)

7.1. Introductory examples

Re-read the slides of §2.5 on anonymous functions and of §2.9 on currying

Example 1: numerical integration

Computation of $\int_a^b f(x)\,dx$ by the trapezoïdal rule:



Specification

function integrate (f, a, b, n)
TYPE: (real $\longrightarrow$ real) $*$ real $*$ real $*$ int $\longrightarrow$ real
PRE: (none)
POST: $\displaystyle\int_a^b f(x)\ dx$, using the trapezoïdal rule with n intervals

Program (`integrate.sml`)

```
fun integrate (f,a,b,n) =
   if n <= 0 orelse b <= a
   then 0.0
   else let val h = (b-a) / real n
        in h * ( f(a) + f(a+h) ) / 2.0 + integrate (f,a+h,b,n-1)
        end
```

```
- fun cube x:real = x * x * x ;
  val cube = fn : real -> real
- integrate ( cube , 0.0 , 2.0 , 10 ) ;
  val it = 4.04 : real
- integrate ( (fn x => x * x * x) , 0.0 , 2.0 , 1000 ) ;
  val it = 4.000004 : real
```

Example 2: image sum

Specification

function sumFct f n
TYPE: (int $\longrightarrow$ int) $\longrightarrow$ int $\longrightarrow$ int
PRE: n $\geq$ 0
POST: f(0) $+$ f(1) $+ \cdots +$ f(n)

Program (`sumImage.sml`)

```
fun sumFct f 0 = f 0
  | sumFct f n = sumFct f (n-1) + f n
```

```
- sumFct (fn x => x * x) 3 ;
  val it = 14 : int
```

Remarks

Higher-order functions manipulate other functions:

- This is *hard* to do in an imperative programming language
- This is a *very* powerful mechanism

7.2. Functional arguments

Example 1: sorting (`mergeSort.sml`)

function sort X
TYPE: int list $\longrightarrow$ int list
PRE: (none)
POST: a non-decreasingly sorted permutation of X

Question: How to sort a list of elements of type α?
Answer: By adding a functional argument,
namely a comparison operator for elements of type α!

function sort order X
TYPE: ($\alpha * \alpha \longrightarrow$ bool) $\longrightarrow \alpha$ list $\longrightarrow \alpha$ list
PRE: (none)
POST: a permutation of X that is 'non-decreasingly' sorted according to order

```
fun  sort order [ ] = [ ]
  |  sort order [x] = [x]
  |  sort order xs =
       let fun merge [ ] M = M
             | merge L [ ] = L
             | merge (L as x::xs) (M as y::ys) =
                 if order (x,y) then x::merge xs M else y::merge L ys
           val (ys,zs) = split xs
       in merge (sort order ys) (sort order zs) end
```

```
-    sort (op >) [5.1, 3.4, 7.4, 0.3, 4.0] ;
  val it = [7.4,5.1,4.0,3.4,0.3] : real list
```

Example 2: binary search trees (`bsTree1.sml`)

Realisation of binary search trees by the bsTree ADT of slide 6.29: limitation to keys of type *integer*

The introduction of a functional argument (say less) of type

$\alpha^= \, * \, \alpha^= \, \rightarrow$ bool

enables the realisation of this ADT with keys of type $\alpha^=$ via

```
datatype (''a,'b) bsTree =
         Void | Bst of (''a * 'b) * (''a,'b) bsTree * (''a,'b) bsTree
```

For instance:

function exists less k T

TYPE: $(\alpha^= * \alpha^= \rightarrow$ bool$) \rightarrow \alpha^= \rightarrow (\alpha^=, \beta)$ bsTree $\rightarrow$ bool

PRE: (none)

POST: **true** if T contains a node with key k under order less
false otherwise

```
fun exists less k Void = false
  | exists less k (Bt((key,s),L,R)) =
      if k = key then true
      else if less (k,key) then exists less k L
      else (* k '>' key *) exists less k R
```

Exercise

- Specify and realise the retrieve, insert, and delete functions by introducing a functional argument less

7.3. Functional abstract datatypes

Consider again the previous specification:

function exists less k T
TYPE: $(\alpha^= * \alpha^= \rightarrow \text{bool}) \rightarrow \alpha^= \rightarrow (\alpha^=, \beta)$ bsTree $\rightarrow$ bool
PRE: (none)
POST: **true** if T contains a node with key k under order less
false otherwise

- The functional argument less must be passed at *each* call
- The less order is not *global* to the binary search tree

Generic abstract datatypes (`bsTree2.sml`)

Introduction of a functional argument of type

$$\alpha^= \; * \; \alpha^= \; \rightarrow \; \text{bool}$$

into the declaration of an ordBsTree datatype:

datatype (''a,'b) bsTree =
Void | Bst **of** (''a * 'b) * (''a,'b) bsTree * (''a,'b) bsTree
type ''a ordering = ''a * ''a -> bool
datatype (''a,'b) ordBsTree = OrdBsTree **of** ''a ordering * (''a,'b) bsTree

```
fun emptyOrdBsTree less = OrdBsTree (less,Void)

fun exists k (OrdBsTree (less,t)) =
   let fun existsAux Void = false
         | existsAux (Bst((key,s),L,R)) =
              if k = key then true
              else if less (k,key) then existsAux L
              else (* k '>' key *) existsAux R
   in existsAux t
   end

fun insert (k,sat) (OrdBsTree (less,t)) = ...

...
```

```
-  val t1 = insert (1,"FP") (emptyOrdBsTree (op <=)) ;
  val t1 = OrdBsTree (fn,Bst((1,"FP"),Void,Void)) :
           (int,string) ordBsTree
-  val OrdBsTree (less,t) = t1 ;
  val less = fn : int ordering
  val t = Bst((1,"FP"),Void,Void) : (int,string) bsTree
-  val test1 = exists 1 t1 ;
  val test1 = true : bool
-  val test2 = exists 2 t1 ;
  val test2 = false : bool
```

7.4. Higher-order functions on lists, etc.

The map function (map.sml)

function map f L
TYPE: ($\alpha \to \beta$) $\to$ α list $\to$ β list
PRE: (none)
POST: $[f(a_1), f(a_2), \ldots, f(a_n)]$ for L = $[a_1, a_2, \ldots, a_n]$

```
fun map f [ ] = [ ]
  | map f (a::As) = f a :: map f As
```

There is a superfluous passing of f at each recursive call, so:

```
fun map f L =
   let fun mapAux [ ] = [ ]
         | mapAux (a::As) = f a :: mapAux As
   in mapAux L
   end
```

```
- fun square x = x * x ;
  val square = fn : int -> int
- map square [1,2,3] ;
  val it = [1,4,9] : int list
- map (fn x => x * x) [1,2,3] ;
  val it = [1,4,9] : int list
- map (fn x => if x < 0 then 0 else x) [~1,2,~3,4,2] ;
  val it = [0,2,0,4,2] : int list
```

The reduce function (reduce.sml)

function reduce f L
TYPE: $(\alpha * \alpha \rightarrow \alpha) \rightarrow \alpha$ list $\rightarrow \alpha$
PRE: L is non-empty
POST: $a_1 \; f \; a_2 \; f \; \cdots \; a_{n-1} \; f \; a_n$ for L = $[a_1, a_2, \ldots, a_n]$,
where f is here used in an infix, right-associating way

```
fun  reduce f [ ] = error "reduce: empty list"
  |  reduce f [a] = a
  |  reduce f (a::As) = f (a, reduce f As)
```

Example 1

```
-   reduce (op +) [1,2,3] ;
  val it = 6 : int
```

Example 2

```
fun  mystery n =
   reduce (fn (a,b) => a andalso b) (map (fn x => n mod x <> 0)
                                         (fromTo 2 (n−1)))
```

Example 3

Computation of $\sum_{1 \leq i \leq n} \sqrt{i}$ via functional composition:

```
infix 3 o
fun  (f o g) x = f (g x)

fun  sumSqrt n = reduce (op +) (map (Math.sqrt o real) (fromTo 1 n))
```

Example 4

```
fun max xs = reduce (fn (x:real,y) => if x > y then x else y) xs
```

Higher-order functions often enable *non-recursive* programs

Example 5

Let L = $[a_1, a_2, \ldots, a_n]$ be a list of at least two real numbers;

the *variance* of L is: $$\frac{\left(\sum_{1 \le i \le n} a_i^2\right)}{n-1} - \frac{\left(\sum_{1 \le i \le n} a_i\right)^2}{n(n-1)}$$

```
fun variance xs =
  let val n = length xs
  in if n <= 1 then error "variance: insufficient amount of data"
     else (reduce (op +) (map square xs)) / ( real (n−1) )
          − square (reduce (op +) xs) / ( real (n ∗ (n−1)) ) end
```

The foldr function

function foldr f e L

TYPE: $(\alpha * \beta \rightarrow \beta) \rightarrow \beta \rightarrow \alpha$ list $\rightarrow \beta$

PRE: (none)

POST: $a_1\ f\ a_2\ f\ \ldots\ a_{n-1}\ f\ a_n\ f$ e for L = $[a_1, a_2, \ldots, a_n]$,
where f is here used in an infix, right-associating way

Examples

foldr (**op** +) 0 $[a_1, \ldots, a_n]$ computes $a_1 + \cdots + a_n +$ 0, for $n \geq 0$

```
fun length xs = foldr (fn (x,n) => n+1) 0 xs
```

The filter function (`filter.sml`)

function filter p L
TYPE: ($\alpha \longrightarrow$ bool) $\longrightarrow \alpha$ list $\longrightarrow \alpha$ list
PRE: (none)
POST: the list of elements of L for which p is true

```
fun  filter p [ ] = [ ]
  |  filter p (x::xs) =
       if  p x then  x :: filter p xs
       else  filter p xs
```

Example

```
-   filter (fn x => x > 0) [~1,2,0,~4,3] ;
  val it = [2,3] : int list
```

The mapbTree function (`mapbTree.sml`)

function mapbTree f t
TYPE: ($\alpha \longrightarrow \beta$) $\longrightarrow \alpha$ bTree $\longrightarrow \beta$ bTree
PRE: (none)
POST: the binary tree t where each node i has been replaced by f(i)

```
fun  mapbTree f Void = Void
  |  mapbTree f (Bt(r,L,R)) =
        Bt(f r, mapbTree f L, mapbTree f R)
```

7.5. Functional tables

A table t of elements with keys of type $\alpha^=$
and satellite data of type β can be seen as a *function*

$$t : \alpha^= \rightarrow \beta$$

We can thus *represent* tables by ML functions!

Inserting an element into a table gives a *new* function:
let v be the table t plus the pair (k, sat):

- $v(i) = t(i) \quad \text{for } i \neq k$
- $v(k) = sat$

The original satellite data for k, if any, thus still exist, in t!

Program (fctTable.sml)

```
-   val emptyTable = (fn i => error "retrieve: non-existing element") ;
  val emptyTable = fn : 'a -> 'b

-   fun insert (k,sat) t = (fn i => if i <> k then t i else sat) ;
  val insert = fn : ''a * 'b -> (''a -> 'b) -> ''a -> 'b

-   val t1 = insert ("Mats",1968) emptyTable ;
  val t1 = fn : string -> int

-   val t2 = insert ("Sten",1937) t1 ;
  val t2 = fn : string -> int

-   t2 "Mats" ;
  val it = 1968 : int
```

Reduction

t2 "Mats"

$\leadsto$ (**fn** i => **if** i <> "Sten" **then** t1 i **else** 1937) "Mats"

$\leadsto$ **if** "Mats" <> "Sten" **then** t1 “Mats” **else** 1937

$\leadsto$ t1 "Mats"

$\leadsto$ (**fn** i => **if** i <> "Mats" **then** emptyTable i **else** 1968) "Mats"

$\leadsto$ **if** "Mats" <> "Mats" **then** emptyTable "Mats" **else** 1968

$\leadsto$ 1968

Assessment

- We can extend a functional table, but *not* shrink it
- The retrieval cost of elements is *high*

7.6. Lazy evaluation

Example

```
fun mult x y =
    if x = 0 then 0
    else x * y
```

Eager evaluation

```
mult (1—1) (3 div 0)
⇝ (fn x => (fn y => if x = 0 then 0 else x * y)) (1—1) (3 div 0)
⇝ (fn x => (fn y => if x = 0 then 0 else x * y)) 0 (3 div 0)
⇝ (fn y => if 0 = 0 then 0 else 0 * y) (3 div 0)
⇝ (fn y => if 0 = 0 then 0 else 0 * y) error
⇝ error
```

- *Value* passing, *eager* evaluation
- Reduce as much as possible *before* applying the function

Lazy evaluation

mult (1−1) (3 **div** 0)
$\leadsto$ (**fn** x => (**fn** y => **if** x = 0 **then** 0 **else** x ∗ y)) (1−1) (3 **div** 0)
$\leadsto$ (**fn** y => **if** (1−1) = 0 **then** 0 **else** (1−1) ∗ y) (3 **div** 0)
$\leadsto$ **if** (1−1) = 0 **then** 0 **else** (1−1) ∗ (3 **div** 0)
$\leadsto$ **if** 0 = 0 **then** 0 **else** (1−1) ∗ (3 **div** 0)
$\leadsto$ 0

- Argument evaluation *as late as possible* (possibly never)
- Evaluation *only when indispensable* for a reduction
- Each argument is evaluated *at most once*
- Lazy evaluation does *not* exist as such in Standard ML, except for the primitives **if then else** , **andalso** , **orelse** , **case of** , and pattern matching
- There *are* lazy ML implementations (LML)

Properties

- If the eager evaluation of expression e gives n_1
  and the lazy evaluation of e gives n_2
  then $n_1 = n_2$
- If the lazy evaluation of e gives $\perp$ (*undefined*)
  then the eager evaluation of e gives $\perp$

Lazy evaluation gives a result *more often*

An ML approximation of lazy evaluation

Approximation of lazy evaluation with functional arguments

Principles

An *actual parameter* a of type α must be 'packaged' into a function, say **(fn** () => a) of type unit $\rightarrow \alpha$

Thus, the parameter a will *not* be evaluated before the call, because the reduced form of (**fn** () => a) is (**fn** () => a) itself Hence there will be fewer execution errors!

When *declaring* a function, if one wants a *formal parameter* p to be "lazy", one now has to write p() instead

Example (lazyMult.sml)

```
fun lazyMult x y =
    if x() = 0 then 0
    else x() * y()
```

The type of lazyMult is (unit → int) → (unit → int) → int

```
-  lazyMult (fn () => 1-1) (fn () => 3 div 0) ;
  val it = 0 : int
```

Reduction

```
lazyMult (fn () => 1—1) (fn () => 3 div 0)
⇝ (fn x => (fn y => if x() = 0 then 0
        else x() ∗ y())) (fn () => 1—1) (fn () => 3 div 0)
⇝ (fn y => if (fn () => 1—1)() = 0 then 0
        else (fn () => 1—1)() ∗ y()) (fn () => 3 div 0)
⇝ if (fn () => 1—1)() = 0 then 0
        else (fn () => 1—1)() ∗ (fn () => 3 div 0)()
⇝ if 0 = 0 then 0 else (fn () => 1—1)() ∗ (fn () => 3 div 0)()
⇝ 0
```

The repeated evaluation of an argument is possible:
this is *not really* lazy evaluation

7.7. Strict and non-strict functions

Consider two functions:

$$h : A \to B$$
$$g : B \to C$$

Reminder: $h(a) = \perp$ denotes that h is not defined on a

The addition of the new mathematical object $\perp$ enables a rigorous formalisation of the behaviour of functions

If $h(a) = \perp$, then what is the value of $g(h(a))$?

Definition

A function g is *strict* iff $g(\perp) = \perp$

Strict functions in ML

```
fun fact 0 = 1
  | fact n = n * fact (n-1)
```

$\forall x \in$ **int**: fact $x = x!$ for $x \geq 0$
fact $x = \perp$ for $x < 0$
fact $\perp = \perp$

The fact function is thus strict

```
fun add x y = x + y
-  add 1 2 ;
   val it = 3 : int
-  add 1 (fact ~2) ;
   ... non-termination ...
```

$\forall x, y \in$ **int**: add $x\ y = x + y$
add $\perp\ y = \perp$
add $x\ \perp = \perp$

The add function is thus strict

Value passing means one can *only* declare *strict* functions!

Non-strict functions in ML

Only some *primitives* of ML are non-strict:

```
-  if true then 1 else fact ~2 ;
   val it = 1 : int
```

These non-strict primitives are necessary:

```
if b = 0 then error "error: division by 0"
          else a div b
```

Lazy evaluation and non-strict functions

Lazy evaluation enables us to declare non-strict functions:

```
fun g x = 5
```

Value passing: g (fact ~2) $= \perp$ $\quad$ (g($\perp$) $= \perp$)
Lazy evaluation: g (fact ~2) $= 5$ $\quad$ (g($\perp$) $\neq \perp$)