

Functional Programming

Shin-Cheng Mu

Academia Sinica and National Taiwan University

2012 Formosan Summer School on
Logic, Language, and Computation
Aug 27 – Sep 7, 2012

Why Functional Programming?

- ▶ “What could be more fashionable than functional programming?”
- ▶ ICFP is having more and more attendees in recent years.
- ▶ A clean model of computation that encourages mathematical reasoning.
- ▶ A “common language” among PL people, upon which other models can be based.

Maximum Segment Sum

- ▶ Given a list of numbers, find the maximum sum of a *consecutive* segment.
 - ▶ $[-1; 3; 3; -4; -1; 4; 2; -1] \Rightarrow 7$
 - ▶ $[-1; 3; 1; -4; -1; 4; 2; -1] \Rightarrow 6$
 - ▶ $[-1; 3; 1; -4; -1; 1; 2; -1] \Rightarrow 4$

Maximum Segment Sum

- ▶ Given a list of numbers, find the maximum sum of a *consecutive* segment.
 - ▶ $[-1; 3; 3; -4; -1; 4; 2; -1] \Rightarrow 7$
 - ▶ $[-1; 3; 1; -4; -1; 4; 2; -1] \Rightarrow 6$
 - ▶ $[-1; 3; 1; -4; -1; 1; 2; -1] \Rightarrow 4$
- ▶ Not trivial. However, there is a linear time algorithm.

Maximum Segment Sum

- ▶ Given a list of numbers, find the maximum sum of a *consecutive* segment.

- ▶ $[-1; 3; 3; -4; -1; 4; 2; -1] \Rightarrow 7$

- ▶ $[-1; 3; 1; -4; -1; 4; 2; -1] \Rightarrow 6$

- ▶ $[-1; 3; 1; -4; -1; 1; 2; -1] \Rightarrow 4$

- ▶

-1	3	1	-4	-1	1	2	-1		
3	4	1	0	2	3	2	0	0	(up + right) ↑ 0
4	4	3	3	3	3	2	0	0	up ↑ right

A Simple Program Whose Proof is Not

- ▶ The specification: $\max \{ \text{sum}(i, j) \mid 0 \leq i \leq j \leq N \}$, where $\text{sum}(i, j) = a[i] + a[i + 1] + \dots + a[j]$.
 - ▶ *What* we want the program to do.
- ▶ The program:

```
s = 0; m = 0;
for (i=0; i<=N; i++) {
    s = max(0, a[j]+s);
    m = max(m, s);
}
```

 - ▶ *How* to do it.
- ▶ They do not look like each other at all!
- ▶ Moral: programs that appear “simple” might not be that simple after all!

Programming and Programming Languages

- ▶ Programming is more than about producing a pile of code.
- ▶ The code has to meet the specification. And, as the example showed, it is sometimes hard to see even for short programs.
- ▶ A good programming languages makes it easier for one to argue for the correctness of a program, by making it easier to reason about programs,
- ▶ which, in my opinion, is the main achievement of functional programming: it is relatively easy to show that a program possess certain properties.
- ▶ We will see in this course (and in this summer school) how programs and proofs are closely related.

This Course

- ▶ Was to be taught by Dr. Tyng-Ruey Chuang, but eventually handed over to me.
- ▶ The functional fragment of the Program Construction course is thus integrated into this course.
- ▶ Program Construction will be mainly about derivation of procedural programs.

Haskell v.s. OCaml

- ▶ OCaml (an implementation of Caml):
 - ▶ Developed and maintained by INRIA since 1985.
 - ▶ A strict language with some impure but useful and practical features — references, exceptions, threads, ...
 - ▶ A relatively mature language with excellent tools and libraries.
 - ▶ In particular, Coq.
- ▶ Haskell:
 - ▶ Developed in late 80's and early 90's.
 - ▶ Non-strict. All impure features are encapsulated by pure constructs (e.g. monads).
 - ▶ A testbed for type theoretic experiments.
 - ▶ Excellent tools for research and rapid prototyping.
- ▶ To prepare for the use of Coq in this summer school, we will be teaching OCaml this year.

Part I

Functions, Values, and Evaluation

Values and Evaluation

Talking to the Interpreter
Evaluation and Termination

Functions

Currying
Sectioning
Definitions

Types — A Brief Start

A Quick Introduction to OCaml

- ▶ We will mostly learn some (boring) syntactical issues, but there are some important messages too.
- ▶ Some contents are inherited from Dr. Tyng-Ruey Chuang's course in the previous year. The rest are adapted from my (Haskell) course notes, which are in turn adapted from Richard Bird's textbook .
- ▶ More OCaml texbooks: Chailloux et. al and Cousineau et. al.
- ▶ Other books on FP and Haskell: Hutton and O'Sullivan et. al.

Simple Expressions

- ▶ Type an expression, and the interpreter evaluates it.

```
# 1+2*3;;
```

```
- : int = 7
```

```
# let pi = 4.0 *. atan 1.0;;
```

```
val pi : float = 3.14159265358979312
```

```
# pi;;
```

```
- : float = 3.14159265358979312
```

- ▶ *int* and *float* are different types.

```
# 1.0 * 2;;
```

```
Error: This expression has type float but an  
expression was expected of type int
```

Defining Functions

- ▶ A function definition:

```
# let square x = x * x;;  
val square : int -> int = <fun>
```

```
# square 3768;;  
- : int = 14197824
```

Defining Functions

Another function definition:

```
# let smaller (x,y) = if x <= y then x else y;;  
val smaller : 'a * 'a -> 'a = <fun>  
  
# smaller (3,4);;  
- : int = 3
```

Evaluation

One possible sequence of evaluating (simplifying, or reducing)
square (3 + 4):

square (3 + 4)

Evaluation

One possible sequence of evaluating (simplifying, or reducing)
square (3 + 4):

$$\begin{aligned}
 & \textit{square} (3 + 4) \\
 = & \quad \{ \textit{definition of } + \} \\
 & \textit{square} 7
 \end{aligned}$$

Evaluation

One possible sequence of evaluating (simplifying, or reducing)
square (3 + 4):

$$\begin{aligned}
 & \textit{square} (3 + 4) \\
 = & \quad \{ \text{definition of } + \} \\
 & \textit{square} 7 \\
 = & \quad \{ \text{definition of } \textit{square} \} \\
 & 7 \times 7
 \end{aligned}$$

Evaluation

One possible sequence of evaluating (simplifying, or reducing)
square (3 + 4):

$$\begin{aligned} & \textit{square} (3 + 4) \\ = & \quad \{ \text{definition of } + \} \\ & \textit{square} 7 \\ = & \quad \{ \text{definition of } \textit{square} \} \\ & 7 \times 7 \\ = & \quad \{ \text{definition of } \times \} \\ & 49 \end{aligned}$$

Another Evaluation Sequence

- ▶ Another possible reduction sequence:

square (3 + 4)

Another Evaluation Sequence

- ▶ Another possible reduction sequence:

$$\begin{aligned}
 & \text{square } (3 + 4) \\
 = & \quad \{ \text{definition of } \textit{square} \} \\
 & (3 + 4) \times (3 + 4)
 \end{aligned}$$

Another Evaluation Sequence

- ▶ Another possible reduction sequence:

$$\begin{aligned}
 & \text{square } (3 + 4) \\
 = & \quad \{ \text{definition of } \textit{square} \} \\
 & (3 + 4) \times (3 + 4) \\
 = & \quad \{ \text{definition of } + \} \\
 & 7 \times (3 + 4)
 \end{aligned}$$

Another Evaluation Sequence

- ▶ Another possible reduction sequence:

$$\begin{aligned}
 & \text{square } (3 + 4) \\
 = & \quad \{ \text{definition of } \textit{square} \} \\
 & (3 + 4) \times (3 + 4) \\
 = & \quad \{ \text{definition of } + \} \\
 & 7 \times (3 + 4) \\
 = & \quad \{ \text{definition of } + \} \\
 & 7 \times 7
 \end{aligned}$$

Another Evaluation Sequence

- ▶ Another possible reduction sequence:

$$\begin{aligned}
 & \text{square } (3 + 4) \\
 = & \quad \{ \text{definition of } \textit{square} \} \\
 & (3 + 4) \times (3 + 4) \\
 = & \quad \{ \text{definition of } + \} \\
 & 7 \times (3 + 4) \\
 = & \quad \{ \text{definition of } + \} \\
 & 7 \times 7 \\
 = & \quad \{ \text{definition of } \times \} \\
 & 49
 \end{aligned}$$

- ▶ In this sequence the rule for *square* is applied first. The final result stays the same.
- ▶ Do different evaluations orders always yield the same thing?

A Non-terminating Reduction

- ▶ Consider the following program:

let *three* $x = 3$

let *infinity* $= \textit{infinity} + 1$

- ▶ Notice how the second definition is different from an assignment $\textit{infinity} := \textit{infinity} + 1$.
- ▶ Try evaluating *three infinity*. If we simplify *infinity* first:

three infinity

$$\begin{aligned}
 &= \{ \text{definition of } \textit{infinity} \} \\
 &\textit{three} (\textit{infinity} + 1) \\
 &= \textit{three} ((\textit{infinity} + 1) + 1) \dots
 \end{aligned}$$

- ▶ If we start with simplifying *three*:

three infinity

$$\begin{aligned}
 &= \{ \text{definition of } \textit{three} \} \\
 &3
 \end{aligned}$$

Values

- ▶ In functional programming, an expression is used solely to describe a *value*.
 - ▶ A value is an abstract being. An expression is but its *representation*.
 - ▶ The value forty-nine can be represented by 49, 7×7 , 110001, or XLIX.
- ▶ The evaluator tries to reduce an expression to a canonical representation. But what canonical representation to use?
 - ▶ We usually go for the *normal form*: the one that cannot be reduced anymore. Thus 49 is preferred over 7×7 .
 - ▶ Some values have a reasonable canonical representation that is not finite. E.g. π .
 - ▶ Some expressions do not have a normal form. E.g. *infinity*.

Evaluation Order

- ▶ There can be many other evaluation orders. As we have seen, some terminates while some do not.
- ▶ A corollary of the **Church–Rosser theorem**: an expression has at most one normal form.
 - ▶ If two evaluation sequences both terminate, they reach the same normal form.
- ▶ Applicative order evaluation: starting with the innermost reducible expression (a redex).
- ▶ Normal order evaluation: starting with the outermost redex.
 - ▶ If an expression has a normal form, normal order evaluation delivers it. Hence the name.
- ▶ For now you can imagine that OCaml uses applicative order evaluation (while Haskell uses normal order evaluation).

Bottom and Strictness

- ▶ Some expressions do not have a normal form. E.g. *infinity*.
- ▶ Some expressions do not denote a well-defined value: $1/0$.
- ▶ To be able to talk about them we denote such undefined values by the symbol \perp .
- ▶ A function f is *strict* if $f \perp = \perp$. Otherwise it is *non-strict*.
- ▶ Functions in OCaml are strict, thus OCaml is called a strict language.
- ▶ Haskell allows non-strict functions.

Values and Evaluation

Talking to the Interpreter

Evaluation and Termination

Functions

Currying

Sectioning

Definitions

Types — A Brief Start

Mathematical Functions

- ▶ Mathematically, a function is a mapping between arguments and results.
 - ▶ A function $f : A \rightarrow B$ maps each element in A to a unique element in B .
- ▶ In contrast, C “functions” are not mathematical functions:
 - ▶ `int y = 1; int f (x:int) { return ((y++) * x); }`
- ▶ *Pure* functional languages exclude such *side-effects*: (unconstrained) assignments, IO, etc.
- ▶ Why removing these useful features? We will talk about that later in this course.

Extensionality

- ▶ Two functions are equal if they give equal results for equal arguments.
 - ▶ $f = g$ if and only if $f\ x = g\ x$ for all x .
- ▶ For instance, $double = double'$:
 - let** $double\ x = x + x$
 - let** $double'\ x = 2 \times x$
 - ▶ They describe different procedures for obtaining the same results.
 - ▶ Being equal means that they can be used interchangeably.
 - ▶ Much of this course is about transformation between functions that are equal.

Pairs

- ▶ Our first compound type: pairs.

```
# let flolac12 = (12,"flolac");;  
val flolac12 : int * string = (12,"flolac")
```

- ▶ **type constructor**: for types a and b , the type $a * b$ can be seen as the collection of stuffs putting one a and one b together.
- ▶ **constructor**: pairs are built by comma ($,$).
- ▶ **deconstructor**: The two components can be extracted respectively by *fst* and *snd*:

```
# fst flolac12;;  
- : int = 12  
# snd flolac12;;  
- : string = "flolac"
```


Currying

- ▶ Consider again the function *smaller*:

```
#let smaller (x, y) = if x ≤ y then x else y;;  
val smaller : 'a * 'a → 'a = ⟨fun⟩
```

- ▶ Another way:

```
#let smallerc x y = if x ≤ y then x else y;;  
val smallerc : 'a → 'a → 'a = ⟨fun⟩
```

- ▶ It's a function returning a function.

```
#smallerc 3;;  
- : int → int = ⟨fun⟩  
#smallerc 3 4;;  
- : int = 3
```

- ▶ Currying: replacing structured arguments by a sequence of simple ones.

Precedence and Association

- ▶ Syntax of some functional languages are tailored toward encouraging the use of curried function.
- ▶ Type: $a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$, not $(a \rightarrow b) \rightarrow c$.
- ▶ Application: $f \ x \ y = (f \ x) \ y$, not $f \ (x \ y)$.
 - ▶ *smallerc 3 4* means *(smallerc 3) 4*.
 - ▶ *square square 3* means *(square square) 3*, which results in a type error.
- ▶ Function application binds tighter than infix operators. E.g. *square 3 + 4* means *(square 3) + 4*.

Why Currying?

- ▶ It exposes more chances to reuse a function, since it can be partially applied.

```
#let twice f x = f (f x);;  
val twice : ('a → 'a) → 'a → 'a = <fun>  
#let quad = twice square;;  
val quad : int → int = <fun>
```

- ▶ Try evaluating *quad* 3:

```
quad 3  
= twice square 3  
= square (square 3)  
= ...
```

Currying and Uncurrying

- ▶ An uncurried function can be converted to a curried function:

```
# let curry f x y = f (x, y)
val curry : ('a * 'b → 'c) → 'a → 'b → 'c = <fun>
```

- ▶ We have *curry smaller = smallerc*.
 - ▶ We will explain the meaning of those types with “dots” later.
- ▶ Can you define *uncurry*?

```
# let uncurry ...
val uncurry : ('a → 'b → 'c) → ('a * 'b → 'c)
```

Sectioning

- ▶ OCaml determines the fixity and associativity of an operator by its first character. See appendix.
- ▶ To use an infix operator in prefix position, surrounded it in parentheses. For example, $(+) 3 4$ is equivalent to $3 + 4$.
- ▶ These operators are curried too. The operator $(+)$ has type $int \rightarrow int \rightarrow int$.
- ▶ Infix operator can be partially applied too.
 - ▶ $(+) 1$ has type $int \rightarrow int$ and increments its argument by one.
 - ▶ $(/.) 1$ has type $float \rightarrow float$. We have $(/.) 1 2 = 1/.2$.

Function Composition

- ▶ Functions composition:

```
# let (<<) f g x = f (g x);;  
val (<<) : ('a → 'b) → ('c → 'a) → 'c → 'b = <fun>
```

- ▶ Putting (\ll) in infix position, we get $(f \ll g) x = f (g x)$.
- ▶ E.g. another way to write *quad*:

```
let quad = square << square
```

- ▶ The identity function: **let** *id* $x = x$.
- ▶ Some important properties:
 - ▶ $id \ll f = f = f \ll id$.
 - ▶ $(f \ll g) \ll h = f \ll (g \ll h)$.

Anonymous Functions

- ▶ A function is a value that can be created, passed around, and used, like any other values.
- ▶ **fun** $x \rightarrow e$ is a function (without a name) that takes an argument x and yields e .
- ▶ Instead of **let** *square* $x = x * x$, one can also write

let *square* = **fun** $x \rightarrow x * x$

- ▶ Such anonymous function is a legacy of λ -calculus, which forms the core theory of functional programming and will be talked about in another course.

Recursive Definitions

- ▶ The keyword **rec** informs that a definition is recursive. E.g.

```
# let rec fact n =  
    if n = 0 then 1 else n * fact (n - 1);;  
val fact : int → int = ⟨fun⟩
```

- ▶ How can something be defined in terms of itself? Are such definitions valid? We will talk about it later.
- ▶ Notice the difference:
 - ▶ **let rec** $f\ x = \dots f \dots$ recursively defines f ;
 - ▶ **let** $f\ x = \dots f \dots$ defines f using some other f defined in the surrounding context.

Mutually Recursive Definitions

- ▶ A group of functions can be defined in terms of each other:

```
# let rec is_even n =  
    if n = 0 then true else is_odd (n - 1)  
and is_odd n =  
    if n = 0 then false else is_even (n - 1);;  
  
val is_even : int → bool  
val is_odd  : int → bool
```

Pattern Matching

- ▶ The famous Fibonacci function:

```
let rec fib n = if n = 0 then 0
                else if n = 1 then 1
                else fib (n - 1) + fib (n - 2)
```

- ▶ A equivalent definition using **match**:

```
let rec fib n = match n with
                | 0 → 0
                | 1 → 1
                | n → fib (n - 1) + fib (n - 2)
```

- ▶ An abbreviation:

```
let rec fib = function
                | 0 → 0
                | 1 → 1
                | n → fib (n - 1) + fib (n - 2)
```

Local Definitions

- ▶ Local bindings are defined by **let ... in ...**, which can be nested:

```
let  $f(x, y) = \mathbf{let} \ a = (x + y)/2 \ \mathbf{in}$   
     $\mathbf{let} \ b = (x + y)/3 \ \mathbf{in}$   
     $(a + 1) * (b + 2)$ 
```

Values and Evaluation

Talking to the Interpreter

Evaluation and Termination

Functions

Currying

Sectioning

Definitions

Types — A Brief Start

Types

- ▶ The universe of values is partitioned into collections, called *types*.
- ▶ Some basic types: *int*, *float*, *bool*, *char*...
- ▶ Type “constructors”: functions, lists, trees ... to be introduced later.
- ▶ Operations on values of a certain type might not make sense for other types. For example: *square square 3*.
- ▶ See appendix for a list of frequently used types and operations.
- ▶ Strong typing: the type of a well-formed expression can be deduced from the constituents of the expression.
 - ▶ It helps you to detect errors.
 - ▶ More importantly, programmers may consider the types for the values being defined before considering the definition themselves, leading to clear and well-structured programs.

Type Inference

- ▶ So far, however, we never had to explicit enter the type of a function. Instead, when OCaml sees a function definition it *infers* its type for us.

```
# let f g x y = g (char_of_int x) + y;;
```

Type Inference

- ▶ So far, however, we never had to explicit enter the type of a function. Instead, when OCaml sees a function definition it *infers* its type for us.

```
# let f g x y = g (char_of_int x) + y;;
val f : (char → int) → int → int → int = <fun>
```

Type Inference

- ▶ So far, however, we never had to explicitly enter the type of a function. Instead, when OCaml sees a function definition it *infers* its type for us.

```
# let f g x y = g (char_of_int x) + y;;  
val f : (char → int) → int → int → int = <fun>
```

- ▶ To make inference possible, OCaml adopts a variation of the *Hindley-Milner type system* — a careful balance of expressiveness and decidability. More on this in the lecture on types.

Polymorphic Types

- ▶ Consider the function

```
# let fork f g x = (f x, g x)
```

```
# fork ((+) 1) ((*) 2) 5;;
```

```
- : int * int = (6, 10)
```

- ▶ Can it have type $(int \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow int \rightarrow int * int$?

Polymorphic Types

- ▶ Consider the function

```
# let fork f g x = (f x, g x)
```

```
# fork ((+) 1) ((*) 2) 5;;
```

```
- : int * int = (6, 10)
```

- ▶ Can it have type $(int \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow int \rightarrow int * int$?
- ▶ What about $(int \rightarrow char) \rightarrow (int \rightarrow string) \rightarrow int \rightarrow char * string$?

Polymorphic Types

- ▶ Consider the function

```
# let fork f g x = (f x, g x)
# fork ((+) 1) ((* 2) 5);;
- : int * int = (6, 10)
```

- ▶ Can it have type $(int \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow int \rightarrow int * int$?
- ▶ What about $(int \rightarrow char) \rightarrow (int \rightarrow string) \rightarrow int \rightarrow char * string$?
- ▶ It turns out that we can assign it a *most general* type:
 $('a \rightarrow 'b) \rightarrow ('a \rightarrow 'c) \rightarrow 'a \rightarrow 'b * 'c$.
- ▶ Types starting with quotes are *type variables*, which can be instantiated to other types.

Polymorphic Types

- ▶ Polymorphic types could be instantiated upon application.

```
# fork int_of_char;;
```

```
- : (char → 'a) → char → int * 'a = <fun>
```

```
# fork char_of_int;;
```

```
- : (int → 'a) → int → char * 'a = <fun>
```

Polymorphism

- ▶ Allowing values of different types to be handled through a uniform interface.
- ▶ Christopher Strachey described two kinds of polymorphism:
- ▶ *Ad-hoc* polymorphism: allowing potentially different code (e.g. printing or comparison for *int* and *float*) to “look the same”.
 - ▶ e.g. function overloading, and method overloading in many OO languages.
 - ▶ e.g. type classes ($Eg\ a \Rightarrow \dots$) in Haskell.
- ▶ *Parametric* polymorphism: allowing the same piece of code, which does not depend on the type of the input data, to be used on a wide range of types.
 - ▶ e.g. *fork*, as we have just seen.
- ▶ In this summer school we will mainly talk about only the second kind.

Summary

- ▶ The functional programming model of computation — given an expression, reduce it to a normal form, if any.
- ▶ Functions are essential building blocks in a functional language They can be applied, composed, passed as arguments, and returned as results.
- ▶ Function definition and pattern matching.
- ▶ Polymorphic types. Types are very important, and we will have a separate course talking about them.

Books and Tutorials

- ▶ E. Chailloux et. al. Developing Applications with Objective Caml..
- ▶ G. Cousineau et. al. The Functional Approach to Programming.
- ▶ R. Bird. Intro. to Functional Programming using Haskell.
- ▶ G. Hutton. Programming in Haskell.
- ▶ B. O'Sullivan et. al. Real World Haskell.

Part II

Induction on Datatypes

User-Defined Types

Induction on Natural Numbers

Datatypes, Functions, and Proofs

What is a Proof, Anyway?

Induction on Lists

Append, and Some of Its Properties

More Inductively Defined Functions

Other Patterns of Induction

VARIANT, OR SUM TYPES

- ▶ A simple example of a sum type:

```
# type state = On | Off;;
```

- ▶ The type *state* has two values

```
# On;;  
- : state = On  
# Off;;  
- : state = Off
```

- ▶ This type is isomorphic to *bool*. One can (and perhaps should) imagine that *bool* is actually defined this way.
- ▶ Value of a sum type can be deconstructed by matching:

```
let flip = function  
  | On → Off  
  | Off → On
```

Option

- ▶ A sum type that carries another type:

```
# type int_option = Nothing | Just of int
```

- ▶ This particular type can be used for handling “exceptions”:

```
# let div x y = if y <> 0 then Just (x/y)  
              else Nothing;;  
val div : int → int → int_option = ⟨fun⟩
```

User-Defined Types

Induction on Natural Numbers

Datatypes, Functions, and Proofs

What is a Proof, Anyway?

Induction on Lists

Append, and Some of Its Properties

More Inductively Defined Functions

Other Patterns of Induction

Proof by Induction on Natural Numbers

- ▶ We've all learnt this principle of proof by induction. To prove that a property P holds for all natural numbers, we show that
 - ▶ $P\ 0$ holds;
 - ▶ $P\ (n + 1)$ holds provided that $P\ n$ does.
- ▶ We can see this as a result of seeing natural numbers as defined by the datatype ¹

type $\mathbb{N} = 0 \mid 1 + \mathbb{N}$

- ▶ That is, any natural number is either 0 , or $1 + n$ where n is a natural number.
- ▶ The type \mathbb{N} is the smallest set such that
 1. 0 is in \mathbb{N} ;
 2. if n is in \mathbb{N} , so is $1 + n$.
- ▶ Thus to show that P holds for all natural numbers, we only need to consider these two cases.

¹Not a real OCaml definition.

Inductively Defined Functions

- ▶ Since the type \mathbb{N} is defined by two cases, it is natural to define functions on \mathbb{N} following the structure:

```
# let rec exp b n = match n with
  | 0 → 1
  | 1+ n → b * exp b n
```

- ▶ Even addition can be defined inductively

```
# let rec m + n = match m with
  | 0 → n
  | 1+ m → 1+ (m + n)
```

- ▶ Exercise: define $(*)$?

Inductively Defined Functions

- ▶ Since the type \mathbb{N} is defined by two cases, it is natural to define functions on \mathbb{N} following the structure:

```
# let rec exp b n = match n with
  | 0 → 1
  | 1+ n → b * exp b n
```

- ▶ Even addition can be defined inductively

```
# let rec (+) m n = match m with
  | 0 → n
  | 1+ m → 1+ (m + n)
```

- ▶ Exercise: define $(*)$?

Embedding \mathbb{N} into *int*

- ▶ Most functional languages do not have a separate type for natural numbers. Instead we have to write:

```
# let rec exp b n = match n with
  | 0 → 1
  | n → b * exp b (n - 1);;
val exp : int → int → int = <fun>
```

- ▶ In this lecture we sometimes use the previous form in proofs. Remember to translate them to “real” programs.

Proof by Induction

To prove properties about \mathbb{N} , we follow the structure as well. E.g. to prove that $b^{m+n} = b^m \times b^n$.

Case $m := 0$:

$$\text{exp } b (0 + n)$$

Proof by Induction

To prove properties about \mathbb{N} , we follow the structure as well. E.g. to prove that $b^{m+n} = b^m \times b^n$.

Case $m := 0$:

$$\begin{aligned} & \text{exp } b (0 + n) \\ = & \quad \{ \text{defn. of } (+) \} \\ & \text{exp } b n \end{aligned}$$

Proof by Induction

To prove properties about \mathbb{N} , we follow the structure as well. E.g. to prove that $b^{m+n} = b^m \times b^n$.

Case $m := 0$:

$$\begin{aligned} & \text{exp } b (0 + n) \\ = & \quad \{ \text{defn. of } (+) \} \\ & \text{exp } b n \\ = & \quad \{ \text{defn. of } (*) \} \\ & 1 * \text{exp } b n \end{aligned}$$

Proof by Induction

To prove properties about \mathbb{N} , we follow the structure as well. E.g. to prove that $b^{m+n} = b^m \times b^n$.

Case $m := 0$:

$$\begin{aligned} & \text{exp } b (0 + n) \\ = & \quad \{ \text{defn. of } (+) \} \\ & \text{exp } b n \\ = & \quad \{ \text{defn. of } (*) \} \\ & 1 * \text{exp } b n \\ = & \quad \{ \text{defn. of exp} \} \\ & \text{exp } b 0 * \text{exp } b n \end{aligned}$$

Proof by Induction

Case $m := 1 + m$:

$$\text{exp } b ((1 + m) + n)$$

Proof by Induction

Case $m := \mathbf{1} + m$:

$$\begin{aligned} & \text{exp } b \ ((\mathbf{1} + m) + n) \\ = & \quad \{ \text{defn. of } (+) \} \\ & \text{exp } b \ (\mathbf{1} + (m + n)) \end{aligned}$$

Proof by Induction

Case $m := 1 + m$:

$$\begin{aligned} & \text{exp } b \ ((1 + m) + n) \\ = & \quad \{ \text{defn. of } (+) \} \\ & \text{exp } b \ (1 + (m + n)) \\ = & \quad \{ \text{defn. of exp} \} \\ & b * \text{exp } b \ (m + n) \end{aligned}$$

Proof by Induction

Case $m := 1 + m$:

$$\begin{aligned} & \text{exp } b \ ((1 + m) + n) \\ = & \quad \{ \text{defn. of } (+) \} \\ & \text{exp } b \ (1 + (m + n)) \\ = & \quad \{ \text{defn. of } \text{exp} \} \\ & b * \text{exp } b \ (m + n) \\ = & \quad \{ \text{induction} \} \\ & b * (\text{exp } b \ m * \text{exp } b \ n) \end{aligned}$$

Proof by Induction

Case $m := 1 + m$:

$$\begin{aligned} & \text{exp } b \ ((1 + m) + n) \\ = & \quad \{ \text{defn. of } (+) \} \\ & \text{exp } b \ (1 + (m + n)) \\ = & \quad \{ \text{defn. of } \text{exp} \} \\ & b * \text{exp } b \ (m + n) \\ = & \quad \{ \text{induction} \} \\ & b * (\text{exp } b \ m * \text{exp } b \ n) \\ = & \quad \{ (*) \text{ associative} \} \\ & (b * \text{exp } b \ m) * \text{exp } b \ n \end{aligned}$$

Proof by Induction

Case $m := \mathbf{1} + m$:

$$\begin{aligned} & \text{exp } b ((\mathbf{1} + m) + n) \\ = & \quad \{ \text{defn. of } (+) \} \\ & \text{exp } b (\mathbf{1} + (m + n)) \\ = & \quad \{ \text{defn. of } \text{exp} \} \\ & b * \text{exp } b (m + n) \\ = & \quad \{ \text{induction} \} \\ & b * (\text{exp } b m * \text{exp } b n) \\ = & \quad \{ (*) \text{ associative} \} \\ & (b * \text{exp } b m) * \text{exp } b n \\ = & \quad \{ \text{defn. of } \text{exp} \} \\ & \text{exp } b (\mathbf{1} + m) * \text{exp } b n \end{aligned}$$

Structure Proofs by Programs

- ▶ The inductive proof could be carried out smoothly, because both $(+)$ and *exp* are defined inductively on \mathbb{N} .
- ▶ The structure of the proof follows the structure of the program, which in turn follows the structure of the datatype the program is defined on.

Lists and Natural Numbers

- ▶ We have yet to prove that (\times) is associative.
- ▶ The proof is quite similar to the proof for associativity of $(@)$, which we will talk about later.
- ▶ In fact, \mathbb{N} and lists are closely related in structure.
- ▶ Most of us are used to think of numbers as atomic and lists as structured data. Neither is necessarily true.
- ▶ For the rest of the course we will demonstrate induction using lists, while taking the properties for \mathbb{N} as given.

But What is a Proof, Anyway?

Xavier Leroy, “How to prove it”:

Proof by example Prove the case $n = 2$ and suggests that it contains most of the ideas of the general proof.

Proof by intimidation ‘Trivial’.

Proof by cumbersome notation Best done with access to at least four alphabets and special symbols.

Proof by reference to inaccessible literature a simple corollary of a theorem to be found in a privately circulated memoir of the Slovenian Philological Society, 1883.

Proof by personal communication ‘Eight-dimensional colored cycle stripping is NP-complete [Karp, personal communication] (in the elevator).’

Proof by appeal to intuition Cloud-shaped drawings.

A semantic proof

A map of London is placed on the ground of Trafalgar Square. There is a point on the map that is directly above the point on the ground that it represents.

Proof.

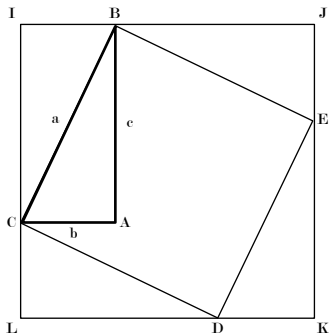
The map is directly above a part of London. Thus the entire map is directly above the part of the area which it represents. Now, the smaller area of the map representing Central London is also above the part of the area which it represents. Within the area representing Central London, Trafalgar Square is marked, and this yet smaller part of the map is directly above the part it represents. Continuing this way, we can find smaller and smaller areas of the map each of which is directly above the part of the area which it represents. In the limit we reduce the area on the map to a single point. □

Proof of Pythagoras's Theorem

Let ABC be a triangle with $\widehat{BAC} = 90^\circ$. Let the lengths of BC , AC , AB be, respectively, a , b , and c . We wish to prove that $a^2 = b^2 + c^2$. Construct a square $IJKL$, of side $b + c$, and a square $BCDE$, of side a . Clearly, $area(IJKL) = (b + c)^2$. But

$$\begin{aligned} area(IJKL) &= area(BCDE) + \\ &\quad 4 \times area(ABC) \\ &= a^2 + abc. \end{aligned}$$

That is, $(b + c)^2 = a^2 + 2bc$,
whence $b^2 + c^2 = a^2$.



Informal v.s. Formal Proofs

- ▶ To read an informal proof, we are expected to have a good understanding of the problem domain, the meaning of the natural language statements, and the language of mathematics.
- ▶ A *formal* proof shifts some of the burdens to the “form”: the symbols, the syntax, and rules manipulating them. “Let the symbols do the work!”
- ▶ Our proof of the swapping program is formal:

$$\{x = A \wedge y = B\}$$

$x := x - y; y := x + y; x := y - x$

$$\{x = B \wedge y = A\}.$$

Tsuru-Kame Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

Tsuru-Kame Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

- ▶ The kindergarten approach: plain simple enumeration!
 - ▶ Crane 0, Tortoise 5 ... No.
 - ▶ Crane 1, Tortoise 4 ... No.
 - ▶ Crane 2, Tortoise 3 ... No.
 - ▶ Crane 3, Tortoise 2 ... **Yes!**
 - ▶ Crane 4, Tortoise 1 ... No.
 - ▶ Crane 5, Tortoise 0 ... No.

Tsuru-Kame Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

- ▶ Elementary school: let's do some reasoning . . .
 - ▶ If all 5 animals were cranes, there ought to be $5 \times 2 = 10$ legs.
 - ▶ However, there are in fact 14 legs. The extra 4 legs must belong to some tortoises. There must be $(14 - 10)/2 = 2$ tortoises.
 - ▶ So there must be $5 - 2 = 3$ cranes.
- ▶ It generalises to larger numbers of heads and legs.
- ▶ Given a different problem, we have to come up with another different way to solve it.

Tsuru-Kame Zan

The Tsuru-Kame Problem

Some cranes (tsuru) and tortoises (kame) are mixed in a cage. Known is that there are 5 heads and 14 legs. Find out the numbers of cranes and tortoises.

- ▶ Junior high school: algebra!

$$x + y = 5$$

$$2x + 4y = 14.$$

- ▶ It's a general approach applicable to many other problems ...
- ▶ ... and perhaps easier.
- ▶ However, it takes efforts to learn!

Formal Proof

Recall our proof

$$\begin{aligned} & \text{exp } b ((1+ m) + n) \\ = & \quad \{ \text{defn. of } (+) \} \\ & \text{exp } b (1 + (m + n)) \\ = & \quad \{ \text{defn. of exp} \} \\ & b \times \text{exp } b (m + n) \\ = & \quad \{ \text{induction} \} \\ & b \times (\text{exp } b m \times \text{exp } b n) \\ = & \quad \{ (\times) \text{ associative} \} \\ & (b \times \text{exp } b m) \times \text{exp } b n \\ = & \quad \{ \text{defn. of exp} \} \\ & \text{exp } b (m + 1) \times \text{exp } b n \end{aligned}$$

It has a rather formal taste.

User-Defined Types

Induction on Natural Numbers

Datatypes, Functions, and Proofs

What is a Proof, Anyway?

Induction on Lists

Append, and Some of Its Properties

More Inductively Defined Functions

Other Patterns of Induction

Lists in OCaml

- ▶ Traditionally an important datatype in functional languages.
- ▶ In OCaml, all elements in a list must be of the same type.

```
# [1; 2; 3; 4];;  
- : int list = [1; 2; 3; 4]
```

- ▶ `[true; false; true]` has type `bool list`.
- ▶ `[[1; 2]; []; [6; 7]]` has type `int list list`.
- ▶ `[(+); (-); (/)]` has type `(int → int → int) list`.
- ▶ `[]`, the empty list, has type `'a list`.

List as a Datatype

- ▶ `[]` is the empty list whose element type is not determined.
- ▶ If a list is non-empty, the leftmost element is called its *head* and the rest its *tail*.
- ▶ The constructor `(::) :: 'a → 'a list → 'a list` builds a list. E.g. in `x :: xs`, `x` is the head and `xs` the tail of the new list.
- ▶ You can think of a list as being defined by

type `'a list = [] | 'a :: 'a list`

- ▶ `[1; 2; 3]` is an abbreviation of `1 :: (2 :: (3 :: []))`.

Head and Tail

- ▶ The following functions are in module *List*. You have to issue a command
- ▶ $hd : 'a\ list \rightarrow 'a$. e.g. $hd\ [1; 2; 3] = 1$.
- ▶ $tl : 'a\ list \rightarrow 'a\ list$. e.g. $tl\ [1; 2; 3] = [2; 3]$.
- ▶ They are both partial functions on non-empty lists (exceptions are raised when applied to empty lists).
- ▶ $length : 'a\ list \rightarrow int$ returns the length of a list.

Finite Lists are Inductively Defined

- ▶ Recall that a (finite) list can be seen as a datatype defined by:²

type *'a list* = [] | 'a :: 'a list

- ▶ Every list is built from the base case [], with elements added by (::) one by one: [1; 2; 3] = 1 :: (2 :: (3 :: [])).
- ▶ The type *'a list* is the smallest set such that
 1. [] is in *'a list*;
 2. if *xs* is in *'a list* and *x* is in *a*, *x :: xs* is in *'a list* as well.
- ▶ Compare with the definition of \mathbb{N} !
- ▶ But what about infinite lists?
 - ▶ For now let's consider finite lists only, as having infinite lists make the *semantics* much more complicated.

²Not a real OCaml definition.

Inductively Defined Functions on Lists

- ▶ Many functions on lists can be defined according to how a list is defined. Eg. summation:

```
# let rec sum = function
  | [] → 0
  | x :: xs → x + sum xs;;
val sum : int list → int = <fun>
```

- ▶ E.g. “mapping” over a list:

```
# let rec map f = function
  | [] → []
  | x :: xs → f x :: map f xs;;
val map : ('a → 'b) → 'a list → 'b list
```

List Append

- ▶ The function (`@`) appends two lists into one ³

```
let rec (@) xs ys = match xs with
  | [] → ys
  | x :: xs → x :: (xs @ ys)
val (@) : 'a list → 'a list → 'a list
```

- ▶ Compare the definition with that of (`+`)!

³This function has an alias: *append*.

Proof by Structural Induction on Lists

- ▶ Recall that every finite list is built from the base case `[]`, with elements added by `(::)` one by one.
- ▶ The type *'a list* is the smallest set such that
 1. `[]` is in *'a list*;
 2. if `xs` is in *'a list* and `x` is in `a`, `x :: xs` is in *'a list* as well.
- ▶ To prove that some property P holds for all finite lists, we show that
 1. $P []$ holds;
 2. $P (x :: xs)$ holds, provided that $P xs$ holds.

Appending is Associative

To prove that $xs @ (ys @ zs) = (xs @ ys) @ zs$. Case $xs := []$:

$$\begin{aligned} & [] @ (ys @ zs) \\ = & \quad \{ \text{defn. of } (@) \} \\ & ys @ zs \\ = & \quad \{ \text{defn. of } (@) \} \\ & ([] @ ys) @ zs \end{aligned}$$

Appending is Associative

Case $xs := x :: xs$:

$$\begin{aligned} & (x :: xs) @ (ys @ zs) \\ = & \quad \{ \text{defn. of } (@) \} \\ & x :: (xs @ (ys @ zs)) \\ = & \quad \{ \text{induction} \} \\ & x :: ((xs @ ys) @ zs) \\ = & \quad \{ \text{defn. of } (@) \} \\ & (x :: (xs @ ys)) @ zs \\ = & \quad \{ \text{defn. of } (@) \} \\ & ((x :: xs) @ ys) @ zs \end{aligned}$$

Length

- ▶ The function *length* can be defined inductively:

```
let rec length = function  
  | [] → 0  
  | x :: xs → 1 + length xs  
val length : 'a list → int
```

- ▶ Exercise: prove that *length* distributes into (*@*):

$$\text{length } (xs @ ys) = \text{length } xs + \text{length } ys$$

Concatenation

- ▶ While $(@)$ repeatedly applies $(::)$, the function *concat* repeatedly calls $(@)$:

```
let rec concat = function  
  | [] → []  
  | xs :: xss → xs @ concat xss  
val concat : 'a list list → 'a list
```

- ▶ Compare with *sum*.
- ▶ Exercise: prove $sum \ll concat = sum \ll map\ sum$.

Why Functional Programming?

- ▶ Back to the question: why functional programming? Why removing useful features (assignment, concurrency, IO...)?
- ▶ By doing so, the functions possess richer mathematical properties we can exploit.
 - ▶ Contrast: in C we cannot even be sure that
$$f(1) + f(1) = 2 * f(1).$$
- ▶ We can prove properties about functions. The properties can be used to optimise programs.
- ▶ These properties even help to construct programs, which will be the subject of another course.

Definition by Induction/Recursion

- ▶ Rather than giving commands, in functional programming we specify values; instead of performing repeated actions, we define values on inductively defined structures.
- ▶ Thus induction (or in general, recursion) is the only “control structure” we have. (We do identify and abstract over plenty of patterns of recursion, though.)
- ▶ To inductively define a function f on lists, we specify a value for the base case ($f []$) and, assuming that $f xs$ has been computed, consider how to construct $f (x :: xs)$ out of $f xs$.

Filter

- ▶ *filter* p xs keeps only those elements in xs that satisfy p .

```
let rec filter p = function
  | [] → []
  | x :: xs → if p x then x :: filter p xs
              else filter p xs
val filter : ('a → bool) → 'a list → 'a list
```

Take and Drop

- ▶ Recall *take* and *drop*, which we used in the previous exercise.

```
let rec take n xs = match n, xs with  
  | 0, _ → []  
  | (1+ n), [] → []  
  | (1+ n), (x :: xs) → x :: take n xs  
val take : int → 'a list → 'a list
```

```
let rec drop n xs = match n, xs with  
  | 0, xs → xs  
  | (1+ n), [] → []  
  | (1+ n), (x :: xs) → drop n xs  
val drop : int → 'a list → 'a list
```

- ▶ Prove: $take\ n\ xs\ @\ drop\ n\ xs = xs$, for all n and xs .

TakeWhile and DropWhile

- ▶ *takeWhile* p xs yields the longest prefix of xs such that p holds for each element.

```
let rec takeWhile p = function
  | [] → []
  | x :: xs → if p x then x :: takeWhile p xs
               else [];
val takeWhile : ('a → bool) → 'a list → 'a list
```

- ▶ *dropWhile* p xs drops the prefix from xs .

```
let rec dropWhile p = function
  | [] → []
  | x :: xs → if p x then dropWhile p xs
               else x :: xs;;
val dropWhile : ('a → bool) → 'a list → 'a list
```

- ▶ Prove: $takeWhile\ p\ xs\ @\ dropWhile\ p\ xs = xs$.

List Reversal

- ▶ $reverse [1; 2; 3; 4] = [4; 3; 2; 1]$.

```
let rec reverse = function  
  | [] → []  
  | x :: xs → reverse xs @ [x];;  
val reverse : 'a list → 'a list
```

All Prefixes and Suffixes

- ▶ Can you define these functions? Both *inits* and *tails* have type $'a \text{ list} \rightarrow 'a \text{ list list}$.
- ▶ *inits* $[1; 2; 3] = [[]; [1]; [1; 2]; [1; 2; 3]]$
- ▶ *tails* $[1; 2; 3] = [[1; 2; 3]; [2; 3]; [3]; []]$

Totality

- ▶ Structure of our definitions so far:

```
let rec  $f$  = function  
  | [] → ...  
  |  $x :: xs$  → ...  $f$   $xs$  ...
```

- ▶ Both the empty and the non-empty cases are covered, guaranteeing there is a matching clause for all inputs.
- ▶ The recursive call is made on a “smaller” argument, guaranteeing termination.
- ▶ Together they guarantee that every input is mapped to some output. Thus they define *total* functions on lists.

Variations with the Base Case

- ▶ Some functions discriminate between several base cases. E.g.

```
let rec fib = function
  | 0 → 0
  | 1 → 1
  | n → fib (n - 1) + fib (n - 2)
```

- ▶ Some functions make more sense when it is defined only on non-empty lists:

```
let rec f = function
  | [x] → ... x ...
  | x :: xs → ... f xs ...
```

- ▶ What about totality?
 - ▶ They can be seen as functions defined on a different datatype:

```
type 'a list+ = [a] | a :: 'a list+
```

- ▶ We do not want to define *map*, *filter* again for *'a list⁺*. Thus we reuse *'a list* and pretend that we were talking about *'a list⁺*.
 - ▶ It's the same with \mathbb{N} . We embedded \mathbb{N} into *Int*.
 - ▶ Ideally we'd like to have some form of *subtyping*. But that makes the type system more complex.

Lexicographic Induction

- ▶ It also occurs often that we perform *lexicographic induction* on multiple arguments: some arguments decrease in size, while others stay the same.
- ▶ E.g. the function *merge* merges two sorted lists into one sorted list:

```
let rec merge xs ys = match xs, ys with
| [], []           → []
| [], y :: ys     → y :: ys
| x :: xs, []     → x :: xs
| x :: xs, y :: ys → if x ≤ y
                       then x :: merge xs (y :: ys)
                       else y :: merge (x :: xs) ys
```

Non-Structural Induction

- ▶ In most of the programs we've seen so far, the recursive calls are made on direct sub-components of the input (e.g. $f(x :: xs) = ..f xs..$). This is called *structural induction*.
 - ▶ It is relatively easy for compilers to recognise structural induction and determine that a program terminates.
- ▶ In fact, we can be sure that a program terminates if the arguments get “smaller” under some (well-founded) ordering.

Mergesort

- ▶ In the implementation of mergesort below, for example, the arguments always get smaller in size.

```
let rec msort = function  
  [] → []  
  [x] → [x]  
  xs → let n = length xs / 2 in  
    let ys = take n xs in  
    let zs = drop n xs in  
    merge (msort ys) (msort zs)
```

- ▶ What if we omit the case for [x]?
- ▶ If all cases are covered, and all recursive calls are applied to smaller arguments, the program defines a total function.

A Non-Terminating Definition

- ▶ Example of a function, where the argument to the recursive does not reduce in size:

```
let rec f = function
  | 0 → 0
  | n → f n
```

- ▶ Certainly f is not a total function. Do such definitions “mean” something? We will talk about these later.

Summary

- ▶ Types may guide you through the design of a program.
 - ▶ Define a datatype inductively.
 - ▶ Define functions by following the structure of the type being processed
 - ▶ Construct proofs by following the structure of the functions whose properties we are concerned with.
- ▶ Equational reasoning: let the symbols do the work!

Part III

Simple Functional Data Structures

On Efficiency of Operations on Lists

Batched Queue

Binary Search Tree

Constant-Time v.s. Linear-Time Operations

- ▶ So far we have (surprisingly) been talking about mathematics without much concern regarding efficiency. Time for a change.
- ▶ Our representation of lists is biased: $(::)$, hd , and tl are constant-time operations, while $init$ and $last$ takes linear-time.

let rec $init$ = function

| $[x] \rightarrow []$

| $x :: xs \rightarrow x :: init\ xs$

- ▶ Consider $init\ [1; 2; 3; 4]$:

Constant-Time v.s. Linear-Time Operations

- ▶ So far we have (surprisingly) been talking about mathematics without much concern regarding efficiency. Time for a change.
- ▶ Our representation of lists is biased: `(::)`, `hd`, and `tl` are constant-time operations, while `init` and `last` takes linear-time.

let rec *init* = function

| [x] → []

| x :: xs → x :: *init* xs

- ▶ Consider `init [1; 2; 3; 4]`:

```
init (1 :: 2 :: 3 :: 4 :: [])  
= 1 :: init (2 :: 3 :: 4 :: [])  
= 1 :: 2 :: init (3 :: 4 :: [])  
= 1 :: 2 :: 3 :: init (4 :: [])  
= 1 :: 2 :: 3 :: []
```

List Concatenation Takes Linear Time

- ▶ Recall (@):

```
let rec (@) xs ys = match xs with  
  | [] → ys  
  | x :: xs → x :: (xs @ ys)  
val (@) : 'a list → 'a list → 'a list
```

List Concatenation Takes Linear Time

- ▶ Recall (@):

```
let rec (@) xs ys = match xs with  
  | [] → ys  
  | x :: xs → x :: (xs @ ys)  
val (@) : 'a list → 'a list → 'a list
```

- ▶ Consider [1; 2; 3] @ [4; 5]:

```
(1 :: 2 :: 3 :: []) @ (4 :: 5 :: [])  
= 1 :: ((2 :: 3 :: []) @ (4 :: 5 :: []))  
= 1 :: 2 :: ((3 :: []) @ (4 :: 5 :: []))  
= 1 :: 2 :: 3 :: ([] @ (4 :: 5 :: []))  
= 1 :: 2 :: 3 :: 4 :: 5 :: []
```

- ▶ (@) runs in time proportional to the length of its left argument.

Sum, Map, etc

- ▶ Functions like *sum*, *maximum*, etc. needs to traverse through the list once to produce a result. So their running time is definitely at least $O(n)$.
- ▶ If f takes time $O(t)$, *map* f takes time $O(n \times t)$ to complete. Similarly with *filter* p .
 - ▶ In a lazy setting, *map* f produces its first result in $O(t)$ time. We won't talk about lazy features for now, however.

Reversing a List

- ▶ The function *reverse* was defined by:

```

let rec reverse = function
  | [] → []
  | x :: xs → reverse xs @ [x];;
val reverse : 'a list → 'a list
    
```

- ▶ E.g.
 $reverse [1; 2; 3; 4] = ((([] @ [4]) @ [3]) @ [2]) @ [1] = [4; 3; 2; 1].$
- ▶ But how about its time complexity? Since ($@$) is $O(n)$, it takes $O(n^2)$ time to revert a list this way.
- ▶ Can we make it faster? Yes, there is a linear time implementation of *reverse*, which will be the subject of the next part. For now, assume that *reverse* is linear.

On Efficiency of Operations on Lists

Batched Queue

Binary Search Tree

Persistence

- ▶ In the world of a functional language, like that of mathematics, values are *persistent* — you have access to all “previous versions” of a value.

```
let x = [1; 2; 3; 4] in  
let x' = reverse x in  
let x'' = tl x' ...
```

Persistency v.s. Efficiency

- ▶ Arrays, when implemented as a consecutive chunk of memory, allows constant time access to an arbitrary element.
- ▶ Such implementation is costly in a world with persistent values:

```
let x = [1; 2; 3; 4] in  
let x' = set x 0 2 in  
let x'' = set x 0 3 ...
```

- ▶ After each assignment do we have to copy the entire array?
- ▶ We thus need some smarter implementations of arrays — and other aggregate data.
- ▶ Still, we often end up having to pay a $O(\log n)$ penalty for persistency.
- ▶ Nevertheless, people have developed some smart data structures for more specific usages.

FILO Queues

- ▶ How do you implement a first-in-last-out queue?
- ▶ If we represent a queue by a list:

```
type 'a queue = 'a list
```

```
let head xs = hd xs
```

```
let tail xs = tl xs
```

```
let snoc xs y = xs@[y]
```

- ▶ The operation *snoc* takes $O(n)$ time.

Representing a Queue by Two Lists

- ▶ Idea: let $([1; 2], [5; 4; 3])$ represent the queue $[1; 2; 3; 4; 5]$.
- ▶ Invariant: the left list is never empty, unless both lists are.

$([1; 2], [5; 4; 3])$

Representing a Queue by Two Lists

- ▶ Idea: let $([1; 2], [5; 4; 3])$ represent the queue $[1; 2; 3; 4; 5]$.
- ▶ Invariant: the left list is never empty, unless both lists are.

$$\begin{array}{l} ([1; 2], [5; 4; 3]) \\ \longrightarrow \quad \{ \text{remove } 1 \} \\ ([2], [5; 4; 3]) \end{array}$$

Representing a Queue by Two Lists

- ▶ Idea: let $([1; 2], [5; 4; 3])$ represent the queue $[1; 2; 3; 4; 5]$.
- ▶ Invariant: the left list is never empty, unless both lists are.

$([1; 2], [5; 4; 3])$
→ $\{ \text{remove } 1 \}$
 $([2], [5; 4; 3])$
→ $\{ \text{add } 6 \}$
 $([2], [6; 5; 4; 3])$

Representing a Queue by Two Lists

- ▶ Idea: let $([1; 2], [5; 4; 3])$ represent the queue $[1; 2; 3; 4; 5]$.
- ▶ Invariant: the left list is never empty, unless both lists are.

$([1; 2], [5; 4; 3])$
→ { remove 1 }
 $([2], [5; 4; 3])$
→ { add 6 }
 $([2], [6; 5; 4; 3])$
→ { remove 2 }
 $([], [6; 5; 4; 3])$

Representing a Queue by Two Lists

- ▶ Idea: let $([1; 2], [5; 4; 3])$ represent the queue $[1; 2; 3; 4; 5]$.
- ▶ Invariant: the left list is never empty, unless both lists are.

$([1; 2], [5; 4; 3])$
→ { remove 1 }
 $([2], [5; 4; 3])$
→ { add 6 }
 $([2], [6; 5; 4; 3])$
→ { remove 2 }
 $([], [6; 5; 4; 3])$
→ { shifting elements }
 $([3; 4; 5; 6], [])$

Representing a Queue by Two Lists

- ▶ Idea: let $([1; 2], [5; 4; 3])$ represent the queue $[1; 2; 3; 4; 5]$.
- ▶ Invariant: the left list is never empty, unless both lists are.

$([1; 2], [5; 4; 3])$
→ { remove 1 }
 $([2], [5; 4; 3])$
→ { add 6 }
 $([2], [6; 5; 4; 3])$
→ { remove 2 }
 $([], [6; 5; 4; 3])$
→ { shifting elements }
 $([3; 4; 5; 6], [])$
→ { remove 3 }
 $([4; 5; 6], [])$

Amortized Constant Time

- ▶ Removal and addition are constant time operations.
- ▶ Shifting is linear in the worst case, but it cannot happen all the time!
- ▶ Each element can be shifted from right to left at most once. Thus the linear cost of one shift can be distributed to each addition.
- ▶ We say that *shift* is an *amortized*-constant time operation.

FIFO Queue: Methods

- ▶ Define **type** `'a queue = 'a list * 'a list`.
- ▶ Goal: define the following methods:
 - ▶ `empty : 'a queue`
 - ▶ `is_empty : 'a queue → bool`
 - ▶ `head : 'a queue → 'a`
 - ▶ `tail : 'a queue → 'a queue`
 - ▶ `snoc : 'a queue → 'a → 'a queue`

Empty Queue

```
type 'a queue = 'a list * 'a list
```

```
let empty = ([], [])
```

```
let is_empty = function
```

```
  | [], _ → true
```

```
  | _ → false
```

Extracting Elements

let *head* = **function**

| $x :: xs, ys \rightarrow x$

let *tail* = **function**

| $x :: xs, ys \rightarrow \text{shift } (xs, ys)$

Shifting and Adding Elements

```
let shift = function  
  | [], ys → (rev ys, [])  
  | q → q
```

```
let snoc (xs, ys) y = shift (xs, y :: ys)
```

On Efficiency of Operations on Lists

Batched Queue

Binary Search Tree

Binary Tree

- ▶ There are many variations of binary trees — internally labelled, externally labelled ...
- ▶ Here we will talk about one particular instance:

```
type 'a tree = Empty
             | Node of 'a tree * 'a * 'a tree
```

- ▶ Example:

```
Node (Node (Empty, 1, Empty),
      2,
      Node (Node (Empty, 3, Empty),
            4,
            Empty))
```

Inductively Defined Functions on Trees

- ▶ Height of a tree:

```
let rec height = function  
  | Empty → 0  
  | Node (t, -, u) →  
    1 + max (height t) (height u)
```

- ▶ Size of a tree:

```
let rec size = function  
  | Empty → 0  
  | Node (t, -, u) → 1 + size t + size u
```

Inductively Defined Functions on Trees

- ▶ Elements of a tree:

let rec flatten = function

| *Empty* → []

| *Node* (*t*, *x*, *u*) →

flatten t @ [*x*] @ *flatten u*

- ▶ It's inorder traversal. Can you define preorder and postorder traversals?
- ▶ All these functions look similar. Is there a more general definition that covers them all?

Binary Search Tree

- ▶ A binary tree with the invariant: in every *Node* (t, x, u) , all elements in t are smaller than x , and all elements in u are greater than x .
- ▶ Define the following methods:
 - ▶ *member* : $'a \rightarrow 'a \text{ tree} \rightarrow \text{bool}$
 - ▶ *insert* : $'a \rightarrow 'a \text{ tree} \rightarrow 'a \text{ tree}$

Membership

```
let rec member x = function  
  | Empty → false  
  | Node (t, y, u) →  
    if x < y then member x t  
    else if y < x then member x u  
    else true
```

Insertion

- ▶ Inserting an element into a tree:

```
let rec insert x = function
  | Empty → Node (Empty, x, Empty)
  | Node (t, y, u) as s →
    if x < y then Node (insert x t, y, u)
    else if y > x then Node (t, y, insert x u)
    else s
```

- ▶ The inserted tree is not balanced. Thus *member* and *insert* are both $O(n)$ in the worst case.
- ▶ One could go for more advanced tree-like data structure for better complexity. There are plenty of them: red-black trees, 2-3 trees ...
- ▶ There has been a whole book about functional data structures! [?]

Summary

- ▶ Values in functional languages are persistent. For that we lose some efficiency.
- ▶ In most cases, we still gain reasonable efficiency by carefully designed data structures.
- ▶ Data structure works under assumptions that certain invariants hold. These invariants are usually implicit in their definitions, and need to be proved separately.

Part IV

Program Calculation

The Unfold/Fold Transformation

Example: Sum of Squares

Accumulating Parameters

Fast List Reversal

Tail Recursion and Loops

Being Quicker by Doing More!

Proof by Strengthening

Tupling

Verification v.s. Derivation

- ▶ Verification: given a program, prove that it is correct with respect to some specification.
- ▶ Derivation: start from the specification, and attempt to construct *only* correct programs!
- ▶ Dijkstra: *“to prove the correctness of a given program, was in a sense putting the cart before the horse. A much more promising approach turned out to be letting correctness proof and program grow hand in hand: with the choice of the structure of the correctness proof one designs a program for which this proof is applicable.”*
- ▶ What happened so far is that theoretical development of one side benefits the other.
- ▶ We focus on verification today, and talk about derivation tomorrow.

Program Derivation

- ▶ Wikipedia: *program derivation* is the derivation of a program from its specification, by mathematical means.
- ▶ To write a formal specification (which could be non-executable), and then apply mathematically correct rules in order to obtain an executable program.
- ▶ The program thus obtained is correct by construction.

A Typical Functional Program Derivation

$max \{ sum (i,j) \mid 0 \leq i \leq j \leq N \}$
= { Premise 1 }
 $max \ll map \ sum \ll concat \ll map \ inits \ll tails$
= { Premise 2 }
...
= { ... }
The final program!

The Unfold/Fold Transformation

Example: Sum of Squares

Accumulating Parameters

Fast List Reversal

Tail Recursion and Loops

Being Quicker by Doing More!

Proof by Strengthening

Tupling

Sum and Map

- ▶ Recall: the function *sum* adds up the numbers in a list:

```
# let rec sum = function
  | [] → 0
  | x :: xs → x + sum xs;;
```

- ▶ E.g. *sum* [7; 9; 11] = 27.
- ▶ The function *map f* takes a list and builds a new list by applying *f* to every item in the input:

```
# let rec map f = function
  | [] → []
  | x :: xs → f x :: map f xs;;
val map : ('a → 'b) → 'a list → 'b list
```

- ▶ E.g. *map square* [3; 4; 6] = [9; 16; 36].

The Unfold/Fold Transformation

Example: Sum of Squares

Accumulating Parameters

Fast List Reversal

Tail Recursion and Loops

Being Quicker by Doing More!

Proof by Strengthening

Tupling

Sum of Squares

- ▶ Given a sequence a_1, a_2, \dots, a_n , compute $a_1^2 + a_2^2 + \dots + a_n^2$.
Specification: **let** *sumsq* = *sum* \ll *map square*.
- ▶ The spec. builds an intermediate list. Can we eliminate it?
- ▶ The input is either empty or not. When it is empty:

sumsq []

Sum of Squares

- ▶ Given a sequence a_1, a_2, \dots, a_n , compute $a_1^2 + a_2^2 + \dots + a_n^2$.
Specification: **let** $sumsq = sum \ll map\ square$.
- ▶ The spec. builds an intermediate list. Can we eliminate it?
- ▶ The input is either empty or not. When it is empty:

$$\begin{aligned} &sumsq [] \\ = &\{ \text{definition of } sumsq \} \\ &(sum \ll map\ square) [] \end{aligned}$$

Sum of Squares

- ▶ Given a sequence a_1, a_2, \dots, a_n , compute $a_1^2 + a_2^2 + \dots + a_n^2$.
Specification: **let** $sumsq = sum \ll map\ square$.
- ▶ The spec. builds an intermediate list. Can we eliminate it?
- ▶ The input is either empty or not. When it is empty:

$$\begin{aligned} &sumsq [] \\ = &\{ \text{definition of } sumsq \} \\ &(sum \ll map\ square) [] \\ = &\{ \text{function composition} \} \\ &sum (map\ square []) \end{aligned}$$

Sum of Squares

- ▶ Given a sequence a_1, a_2, \dots, a_n , compute $a_1^2 + a_2^2 + \dots + a_n^2$.
Specification: **let** $sumsq = sum \ll map\ square$.
- ▶ The spec. builds an intermediate list. Can we eliminate it?
- ▶ The input is either empty or not. When it is empty:

$$\begin{aligned} &sumsq [] \\ = &\{ \text{definition of } sumsq \} \\ &(sum \ll map\ square) [] \\ = &\{ \text{function composition} \} \\ &sum (map\ square []) \\ = &\{ \text{definition of } map \} \\ &sum [] \end{aligned}$$

Sum of Squares

- ▶ Given a sequence a_1, a_2, \dots, a_n , compute $a_1^2 + a_2^2 + \dots + a_n^2$.
Specification: **let** $sumsq = sum \ll map\ square$.
- ▶ The spec. builds an intermediate list. Can we eliminate it?
- ▶ The input is either empty or not. When it is empty:

$$\begin{aligned} &sumsq [] \\ = &\{ \text{definition of } sumsq \} \\ &(sum \ll map\ square) [] \\ = &\{ \text{function composition} \} \\ &sum (map\ square []) \\ = &\{ \text{definition of } map \} \\ &sum [] \\ = &\{ \text{definition of } sum \} \\ &0 \end{aligned}$$

Sum of Squares, the Inductive Case

- ▶ Consider the case when the input is not empty:

sumsq ($x :: xs$)

Sum of Squares, the Inductive Case

- ▶ Consider the case when the input is not empty:

$$\begin{aligned} & \text{sumsq } (x :: xs) \\ = & \{ \text{definition of } \text{sumsq} \} \\ & \text{sum } (\text{map } \text{square } (x :: xs)) \end{aligned}$$

Sum of Squares, the Inductive Case

- ▶ Consider the case when the input is not empty:

$$\begin{aligned} & \text{sumsq } (x :: xs) \\ = & \quad \{ \text{definition of } \text{sumsq} \} \\ & \text{sum } (\text{map } \text{square } (x :: xs)) \\ = & \quad \{ \text{definition of } \text{map} \} \\ & \text{sum } (\text{square } x :: \text{map } \text{square } xs) \end{aligned}$$

Sum of Squares, the Inductive Case

- ▶ Consider the case when the input is not empty:

$$\begin{aligned} & \text{sumsq } (x :: xs) \\ = & \quad \{ \text{definition of } \text{sumsq} \} \\ & \text{sum } (\text{map } \text{square } (x :: xs)) \\ = & \quad \{ \text{definition of } \text{map} \} \\ & \text{sum } (\text{square } x :: \text{map } \text{square } xs) \\ = & \quad \{ \text{definition of } \text{sum} \} \\ & \text{square } x + \text{sum } (\text{map } \text{square } xs) \end{aligned}$$

Sum of Squares, the Inductive Case

- ▶ Consider the case when the input is not empty:

$$\begin{aligned} & \text{sumsq } (x :: xs) \\ = & \quad \{ \text{definition of } \text{sumsq} \} \\ & \text{sum } (\text{map } \text{square } (x :: xs)) \\ = & \quad \{ \text{definition of } \text{map} \} \\ & \text{sum } (\text{square } x :: \text{map } \text{square } xs) \\ = & \quad \{ \text{definition of } \text{sum} \} \\ & \text{square } x + \text{sum } (\text{map } \text{square } xs) \\ = & \quad \{ \text{definition of } \text{sumsq} \} \\ & \text{square } x + \text{sumsq } xs \end{aligned}$$

An Alternative Definition for *sumsq*

- ▶ From $sumsq = sum \ll map\ square$, we have proved that

```
let sumsq = function
  | [] → 0
  | x :: xs → square x + sumsq xs
```

- ▶ Equivalently, we have shown that $sum \ll map\ square$ is a solution of

```
f [] = 0
f (x :: xs) = square x + f xs
```

- ▶ However, the solution of the equations above is unique.
- ▶ Thus we can take it as another definition of *sumsq*.
Denotationally it is the same function; operationally, it is (slightly) quicker.

Unfold/Fold Transformation

- ▶ Perhaps the most intuitive, yet still handy, style of functional program derivation.
- ▶ Keep unfolding the definition of functions, apply necessary rules, and finally fold the definition back.
- ▶ It works under the assumption that a function satisfying the derived equations *is* the function defined by the equations.
- ▶ Do not confuse “fold” and “unfold” with *foldr* and *unfoldr*, which are important operations on datatypes and unfortunately cannot be covered in this course.

Reversing a List

- ▶ The function *reverse* was defined by:

```
let rec reverse = function  
  | [] → []  
  | x :: xs → reverse xs @ [x];;  
val reverse : 'a list → 'a list
```

- ▶ E.g.
 $reverse [1; 2; 3; 4] = ((([] @ [4]) @ [3]) @ [2]) @ [1] = [4; 3; 2; 1].$
- ▶ But how about its time complexity? Since $(@)$ is $O(n)$, it takes $O(n^2)$ time to revert a list this way.
- ▶ Can we make it faster?

Introducing an Accumulating Parameter

- ▶ Let us consider a generalisation of *reverse*. Define:

```
let revcat xs ys = reverse xs @ ys
val revcat : 'a list → 'a list → 'a list
```

- ▶ If we can construct a fast implementation of *revcat*, we can implement *reverse* by:

```
let reverse xs = revcat xs []
```

Reversing a List, Base Case

Let us use our old trick. Consider the case when *xs* is []:

revcat [] *ys*

Reversing a List, Base Case

Let us use our old trick. Consider the case when *xs* is []:

$$\begin{aligned} & \text{revcat [] } ys \\ = & \quad \{ \text{definition of revcat} \} \\ & \text{reverse [] @ } ys \end{aligned}$$

Reversing a List, Base Case

Let us use our old trick. Consider the case when xs is `[]`:

$$\begin{aligned} & \text{revcat [] } ys \\ = & \quad \{ \text{definition of } \text{revcat} \} \\ & \text{reverse [] @ } ys \\ = & \quad \{ \text{definition of } \text{reverse} \} \\ & \text{[] @ } ys \end{aligned}$$

Reversing a List, Base Case

Let us use our old trick. Consider the case when xs is `[]`:

$$\begin{aligned} & \text{revcat [] } ys \\ = & \quad \{ \text{definition of } \text{revcat} \} \\ & \text{reverse [] @ } ys \\ = & \quad \{ \text{definition of } \text{reverse} \} \\ & \text{[] @ } ys \\ = & \quad \{ \text{definition of } (@) \} \\ & ys. \end{aligned}$$

Reversing a List, Inductive Case

Case $x :: xs$:

revcat ($x :: xs$) *ys*

Reversing a List, Inductive Case

Case $x :: xs$:

$$\begin{aligned} & \text{revcat } (x :: xs) \text{ } ys \\ = & \quad \{ \text{definition of } \text{revcat} \} \\ & \text{reverse } (x :: xs) @ ys \end{aligned}$$

Reversing a List, Inductive Case

Case $x :: xs$:

$$\begin{aligned} & \text{revcat } (x :: xs) \text{ } ys \\ = & \quad \{ \text{definition of } \text{revcat} \} \\ & \text{reverse } (x :: xs) @ ys \\ = & \quad \{ \text{definition of } \text{reverse} \} \\ & (\text{reverse } xs @ [x]) @ ys \end{aligned}$$

Reversing a List, Inductive Case

Case $x :: xs$:

$$\begin{aligned} & \text{revcat } (x :: xs) \text{ } ys \\ = & \quad \{ \text{definition of revcat} \} \\ & \text{reverse } (x :: xs) @ ys \\ = & \quad \{ \text{definition of reverse} \} \\ & (\text{reverse } xs @ [x]) @ ys \\ = & \quad \{ \text{since } (xs @ ys) @ zs = xs @ (ys @ zs) \} \\ & \text{reverse } xs @ ([x] @ ys) \end{aligned}$$

Reversing a List, Inductive Case

Case $x :: xs$:

$$\begin{aligned} & \text{revcat } (x :: xs) \text{ } ys \\ = & \quad \{ \text{definition of } \text{revcat} \} \\ & \text{reverse } (x :: xs) @ ys \\ = & \quad \{ \text{definition of } \text{reverse} \} \\ & (\text{reverse } xs @ [x]) @ ys \\ = & \quad \{ \text{since } (xs @ ys) @ zs = xs @ (ys @ zs) \} \\ & \text{reverse } xs @ ([x] @ ys) \\ = & \quad \{ \text{definition of } \text{revcat} \} \\ & \text{revcat } xs \ (x :: ys). \end{aligned}$$

Linear-Time List Reversal

- ▶ We have therefore constructed an implementation of *revcat* which runs in linear time!

```
let rec revcat xs ys = match xs with  
| [] → ys  
| x :: xs → revcat xs (x :: ys)
```

- ▶ A generalisation of *reverse* is easier to implement than *reverse* itself? How come?
- ▶ If you try to understand *revcat* operationally, it is not difficult to see how it works.
 - ▶ The partially reverted list is *accumulated* in *ys*.
 - ▶ The initial value of *ys* is set by *reverse xs = revcat xs []*.
 - ▶ Hmm... it is like a *loop*, isn't it?

Tracing Reverse

```
reverse [1; 2; 3; 4]
= revcat [1; 2; 3; 4] []
= revcat [2; 3; 4] [1]
= revcat [3; 4] [2; 1]
= revcat [4] [3; 2; 1]
= revcat [] [4; 3; 2; 1]
= [4; 3; 2; 1]
```

```
let reverse xs = revcat xs []
let rec revcat xs ys = match xs with
| [] → ys
| x :: xs → revcat xs (x :: ys)

xs, ys ← XS, [];
while xs ≠ [] do
  xs, ys ← (tl xs), (hd xs :: ys);
return ys
```


Tail Recursion

- ▶ Tail recursion: a special case of recursion in which the last operation is the recursive call.

$$f\ x_1 \ \dots \ x_n = \{\text{base case}\}$$

$$f\ x_1 \ \dots \ x_n = f\ x'_1 \ \dots \ x'_n$$

- ▶ To implement general recursion, we need to keep a stack of return addresses. For tail recursion, we do not need such a stack.
- ▶ Tail recursive definitions are like loops. Each x_i is updated to x'_i in the next iteration of the loop.
- ▶ The first call to f sets up the initial values of each x_i .

Accumulating Parameters

- ▶ To efficiently perform a computation (e.g. *reverse xs*), we introduce a generalisation with an extra parameter, e.g.:

revcat xs ys = reverse xs @ ys.

- ▶ Try to derive an efficient implementation of the generalised function. The extra parameter is usually used to “accumulate” some results, hence the name.
 - ▶ To make the accumulation work, we usually need some kind of associativity.
- ▶ A technique useful for, but not limited to, constructing tail-recursive definition of functions.

Accumulating Parameter: Another Example

- ▶ Recall the “sum of squares” problem:

```
let sumsq = function  
  | [] → 0  
  | x :: xs → square x + sumsq xs
```

- ▶ The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- ▶ Introduce **let** *ssp xs n* = .
- ▶ Initialisation: **let** *sumsq xs* = .
- ▶ Construct *ssp*:

Accumulating Parameter: Another Example

- ▶ Recall the “sum of squares” problem:

```
let sumsq = function  
  | [] → 0  
  | x :: xs → square x + sumsq xs
```

- ▶ The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- ▶ Introduce **let** *ssp xs n = sumsq xs + n*.
- ▶ Initialisation: **let** *sumsq xs =* `ssp xs 0` **.**
- ▶ Construct *ssp*:

Accumulating Parameter: Another Example

- ▶ Recall the “sum of squares” problem:

```
let sumsq = function  
  | [] → 0  
  | x :: xs → square x + sumsq xs
```

- ▶ The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- ▶ Introduce **let** *ssp xs n* = *sumsq xs* + *n*.
- ▶ Initialisation: **let** *sumsq xs* = *ssp xs* 0.
- ▶ Construct *ssp*:

Accumulating Parameter: Another Example

- ▶ Recall the “sum of squares” problem:

```
let sumsq = function  
  | [] → 0  
  | x :: xs → square x + sumsq xs
```

- ▶ The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- ▶ Introduce **let** *ssp* *xs* *n* = *sumsq* *xs* + *n*.
- ▶ Initialisation: **let** *sumsq* *xs* = *ssp* *xs* 0.
- ▶ Construct *ssp*:

$$\begin{aligned} \textit{ssp} [] n &= 0 + n = n \\ \textit{ssp} (x :: xs) n &= (\textit{square} x + \textit{sumsq} xs) + n \\ &= \textit{sumsq} xs + (\textit{square} x + n) \\ &= \textit{ssp} xs (\textit{square} x + n). \end{aligned}$$

Being Quicker by Doing More?

- ▶ A more generalised program can be implemented more efficiently?
 - ▶ A common phenomena! Sometimes the less general function cannot be implemented inductively at all!
 - ▶ It also often happens that a theorem needs to be generalised to be proved. We will see that later.
- ▶ An obvious question: how do we know what generalisation to pick?
- ▶ There is no easy answer — finding the right generalisation one of the most difficult act in programming!
- ▶ For the past few examples, we choose the generalisation to exploit associativity.
- ▶ Sometimes we simply generalise by examining the form of the formula.

Combine, or Zip

- ▶ A useful function, called *zip* in Haskell:

```
let rec combine xs ys = match xs, ys with  
  | [], [] → []  
  | x :: xs, y :: ys → (x, y) :: combine xs ys  
val combine : 'a list → 'b list → ('a * 'b) list
```

- ▶ E.g. `combine [1;3;5] ['a'; 'b'; 'c'] = [(1, 'a'); (3, 'b'); (5, 'c')]`.

Labelling a List

- ▶ Consider the task of labelling elements in a list with its index.

```
let index xs = combine (fromTo 0 (length xs)) xs  
val index : 'a list → (int * 'a) list
```

- ▶ E.g. `index ['a'; 'b'; 'c'] = [(0, 'a'); (1, 'b'); (2, 'c')]`.

Labelling a List, Inductive Case

- ▶
- ▶ To construct an inductive definition, the case for `[]` is easy.
For the `x :: xs` case:

$$\begin{aligned} & \textit{index} (x :: xs) \\ &= \textit{combine} (\textit{fromTo} 0 (\textit{length} (x :: xs))) (x :: xs) \\ &= \textit{combine} (\textit{fromTo} 0 (1 + \textit{length} xs)) (x :: xs) \\ &= \textit{combine} (0 :: \textit{fromTo} 1 (\textit{length} xs)) (x :: xs) \\ &= (0, x) : \textit{combine} (\textit{fromTo} 1 (\textit{length} xs)) xs \end{aligned}$$

- ▶ Alas, the last line cannot be folded back to *index*!
- ▶ What if we turn the varying part into... a variable?

Labelling a List, Second Attempt

- ▶ Generalise *index*:

```
let idxFrom xs n =  
    combine (fromTo n (n + length xs)) xs
```

- ▶ Initialisation: `let index xs =` .

Labelling a List, Second Attempt

- ▶ Generalise *index*:

```
let idxFrom xs n =  
    combine (fromTo n (n + length xs)) xs
```

- ▶ Initialisation: **let** *index* *xs* = *idxFrom* *xs* 0.

Labelling a List, Second Attempt

- ▶ Generalise *index*:

```
let idxFrom xs n =  
    combine (fromTo n (n + length xs)) xs
```

- ▶ Initialisation: `let index xs = idxFrom xs 0.`
- ▶ We reason:

```
idxFrom (x :: xs) n  
= combine (fromTo n (n + len (x :: xs))) (x :: xs)  
= combine (fromTo n (1 + n + len xs)) (x :: xs)  
= combine (n :: fromTo (1 + n) (1 + n + len xs)) (x :: xs)  
= (n, x) : combine (fromTo (1 + n) (len xs)) xs  
= (n, x) : idxFrom xs (1 + n)
```

Labelling a List, Second Attempt

```
let index xs = idxFrom xs 0  
let idxFrom xs n = match xs with  
  | [] → []  
  | x :: xs → (n, x) : idxFrom xs (1 + n)
```

Summing Up a List in Reverse

- ▶ Prove: $sum \llcorner reverse = sum$, using the definition $reverse\ xs = revcat\ xs\ []$. That is, proving $sum\ (revcat\ xs\ []) = sum\ xs$.

- ▶ Base case trivial. For the case $x :: xs$:

$$\begin{aligned} & sum\ (reverse\ (x :: xs)) \\ &= sum\ (revcat\ (x :: xs)\ []) \\ &= sum\ (revcat\ xs\ [x]) \end{aligned}$$

- ▶ Then we are stuck, since we cannot use the induction hypothesis $sum\ (revcat\ xs\ []) = sum\ xs$.
- ▶ Again, generalise $[x]$ to a variable.

Summing Up a List in Reverse, Second Attempt

- ▶ Second attempt: prove a lemma:

$$\text{sum} (\text{revcat } xs \ ys) =$$

- ▶ By letting $ys = []$ we get the previous property.

Summing Up a List in Reverse, Second Attempt

- ▶ Second attempt: prove a lemma:

$$\text{sum} (\text{revcat } xs \ ys) = \text{sum } xs + \text{sum } ys$$

- ▶ By letting $ys = []$ we get the previous property.

Summing Up a List in Reverse, Second Attempt

- ▶ Second attempt: prove a lemma:

$$\text{sum} (\text{revcat } xs \ ys) = \text{sum } xs + \text{sum } ys$$

- ▶ By letting $ys = []$ we get the previous property.
- ▶ For the case $x :: xs$ we reason:

$$\text{sum} (\text{revcat } (x :: xs) \ ys)$$

Summing Up a List in Reverse, Second Attempt

- ▶ Second attempt: prove a lemma:

$$\text{sum } (\text{revcat } xs \ ys) = \text{sum } xs + \text{sum } ys$$

- ▶ By letting $ys = []$ we get the previous property.
- ▶ For the case $x :: xs$ we reason:

$$\begin{aligned} & \text{sum } (\text{revcat } (x :: xs) \ ys) \\ = & \text{sum } (\text{revcat } xs \ (x :: ys)) \end{aligned}$$

Summing Up a List in Reverse, Second Attempt

- ▶ Second attempt: prove a lemma:

$$\text{sum (revcat } xs \text{ } ys) = \text{sum } xs + \text{sum } ys$$

- ▶ By letting $ys = []$ we get the previous property.
- ▶ For the case $x :: xs$ we reason:

$$\begin{aligned} & \text{sum (revcat (x :: xs) ys)} \\ = & \text{sum (revcat xs (x :: ys))} \\ = & \left\{ \text{induction hypothesis} \right\} \\ & \text{sum } xs + \text{sum (x :: ys)} \end{aligned}$$

Summing Up a List in Reverse, Second Attempt

- ▶ Second attempt: prove a lemma:

$$\text{sum} (\text{revcat } xs \ ys) = \text{sum } xs + \text{sum } ys$$

- ▶ By letting $ys = []$ we get the previous property.
- ▶ For the case $x :: xs$ we reason:

$$\begin{aligned} & \text{sum} (\text{revcat } (x :: xs) \ ys) \\ = & \text{sum} (\text{revcat } xs \ (x :: ys)) \\ = & \{ \text{induction hypothesis} \} \\ & \text{sum } xs + \text{sum } (x :: ys) \\ = & \text{sum } xs + x + \text{sum } ys \\ = & \text{sum } (x : xs) + \text{sum } ys \end{aligned}$$

Work Less by Proving More

- ▶ A stronger theorem is easier to prove! Why is that?
- ▶ By strengthening the theorem, we also have a stronger induction hypothesis, which makes an inductive proof possible.
 - ▶ Finding the right generalisation is an art — it's got to be strong enough to help the proof, yet not too strong to be provable.
- ▶ The same with programming. By generalising a function with additional arguments, it is passed more information it may use, thus making an inductive definition possible.
 - ▶ The speeding up of *revcat*, in retrospect, is an accidental “side effect” — *revcat*, being inductive, goes through the list only once, and is therefore quicker.

A Real Case

- ▶ A property I actually had to prove for a paper:

$$\begin{aligned} & (\forall n : \dots : \text{take } n \ x \leq_d \ \text{drop } n \ x) \\ & \Rightarrow \text{maximum}(\text{map}((@)z) (\text{inits } x)) \\ & = z \uparrow_d (z @ x) \end{aligned}$$

- ▶ It took me quite a while to construct the right generalisation:

$$\begin{aligned} & (\forall n : \dots : y @ \text{take } n \ x \leq_d \ \text{drop } n \ x) \\ & \Rightarrow z \uparrow_d \text{maximum}(\text{map}((@)(z @ y)) (\text{inits } x)) \\ & = z \uparrow_d (z @ y @ x) \end{aligned}$$

- ▶ In another case I spent a week on the right generalisation. Once the right property is found, the actual proof was done in about 20 minutes.
- ▶ “Someone once described research as ‘finding out something to find out, then finding it out’; the first part is often harder than the second.”

Remark

- ▶ The *sum* \ll *reverse* example is superficial — the same property is much easier to prove using the $O(n^2)$ -time definition of *reverse*.
- ▶ That's one of the reason we defer the discussion about efficiency — to prove properties of a function we sometimes prefer to roll back to a slower version.

Steep Lists

- ▶ A *steep list* is a list in which every element is larger than the sum of those to its right:

```
let rec steep = function  
  | [] → true  
  | x :: xs → steep xs && x > sum xs  
val steep : int list → bool
```

- ▶ The definition above, if executed directly, is an $O(n^2)$ program. Can we do better?
- ▶ Just now we learned to construct a generalised function which takes more input. This time, we try the dual technique: to construct a function returning more results.

Generalise by Returning More

- ▶ Recall that $\text{fst } (a, b) = a$ and $\text{snd } (a, b) = b$.
- ▶ It is hard to quickly compute *steep* alone. But if we define

let *steepsum* *xs* = (*steep* *xs*, *sum* *xs*)

- ▶ and manage to synthesise a quick definition of *steepsum*, we can implement *steep* by $\text{steep} = \text{fst} \ll \text{steepsum}$.
- ▶ We again proceed by case analysis. Trivially,

steepsum [] = (*true*, 0).

Deriving for the Non-Empty Case

For the case for non-empty inputs:

steepsum ($x :: xs$)

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$\begin{aligned} & \textit{steepsum} (x :: xs) \\ = & \{ \text{definition of } \textit{steepsum} \} \\ & (\textit{steep} (x :: xs), \textit{sum} (x :: xs)) \end{aligned}$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$\begin{aligned} & \textit{steepsum} (x :: xs) \\ = & \quad \{ \text{definition of } \textit{steepsum} \} \\ & (\textit{steep} (x :: xs), \textit{sum} (x :: xs)) \\ = & \quad \{ \text{definitions of } \textit{steep} \text{ and } \textit{sum} \} \\ & (\textit{steep} xs \ \&\& \ x > \textit{sum} xs, x + \textit{sum} xs) \end{aligned}$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$\begin{aligned} & \text{steepsum } (x :: xs) \\ = & \quad \{ \text{definition of } \text{steepsum} \} \\ & (\text{steep } (x :: xs), \text{sum } (x :: xs)) \\ = & \quad \{ \text{definitions of } \text{steep} \text{ and } \text{sum} \} \\ & (\text{steep } xs \ \&\& \ x > \text{sum } xs, x + \text{sum } xs) \\ = & \quad \{ \text{extracting sub-expressions involving } xs \} \\ & \mathbf{match} \ \text{steep } xs, \text{sum } xs \ \mathbf{with} \\ & \quad | \ b, y \rightarrow (b \ \&\& \ x > y, x + y) \end{aligned}$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$\begin{aligned} & \textit{steepsum} (x :: xs) \\ = & \quad \{ \text{definition of } \textit{steepsum} \} \\ & (\textit{steep} (x :: xs), \textit{sum} (x :: xs)) \\ = & \quad \{ \text{definitions of } \textit{steep} \text{ and } \textit{sum} \} \\ & (\textit{steep} xs \ \&\& \ x > \textit{sum} xs, x + \textit{sum} xs) \\ = & \quad \{ \text{extracting sub-expressions involving } xs \} \\ & \text{match } \textit{steep} \textit{xs}, \textit{sum} \textit{xs} \text{ with} \\ & \quad | b, y \rightarrow (b \ \&\& \ x > y, x + y) \\ = & \quad \{ \text{definition of } \textit{steepsum} \} \\ & \text{match } \textit{steepsum} \textit{xs} \text{ with} \\ & \quad | b, y \rightarrow (b \ \&\& \ x > y, x + y) \end{aligned}$$

Synthesised Program

- ▶ We have thus come up with a $O(n)$ time program:

```
let steep = fst << steepsum
let steepsum = function
  | [] → (true, 0)
  | x :: xs → match steepsum xs with
    | b, y → (b && x > y, x + y)
```

- ▶ Again we observe the phenomena that a more general function is easier to implement.

How Far Can We Go?

- ▶ We will show in an appendix the entire derivation of the maximum segment sum problem.
- ▶ Bird and de Moor conducted a thorough study of optimisation problems — when there is a greedy algorithm, when it can be solved by dynamic programming, etc.
- ▶ Through calculations, we sometimes discover new algorithms, or variations/improvements of existing algorithms.
- ▶ Certainly, not all problems can be solved by calculation. When they do, we gain better understanding of their natures.
- ▶ More case studies of program calculation are still being published in “Functional Pearl” section of ICFP and Journal of Functional Programming.

Summary

- ▶ A program and its correctness proof can be, and should be developed together.
- ▶ Program calculation is one such methodology. From a specification, we stepwise calculate an algorithm by (in)equational reasoning.