Functional Programming

Shin-Cheng Mu

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 (International Conference on Functional Programming) 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.
  - $[-1; 3; 3; -4; -1; 4; 2; -1] \quad \Rightarrow \quad 7$
  - $[-1; 3; 1; -4; -1; 4; 2; -1] \quad \Rightarrow \quad 6$
  - $[-1; 3; 1; -4; -1; 1; 2; -1] \quad \Rightarrow \quad 4$
- Not trivial. However, there is a linear time algorithm.
- $$\begin{array}{cccccccccl} -1 & 3 & 1 & -4 & -1 & 1 & 2 & -1 & & \\ 3 & 4 & 1 & 0 & 2 & 3 & 2 & 0 & 0 & (up + right) \uparrow 0 \\ 4 & 4 & 3 & 3 & 3 & 3 & 2 & 0 & 0 & up \uparrow right \end{array}$$

A Simple Program Whose Proof is Not

- The specification: $max \ \{ \ sum \ (i,j) \mid 0 \leq i \leq j \leq N \ \}$, where $sum \ (i,j) = a[i] + a[i+1] + \ldots + a[i]$.
  - *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

1 Values and Evaluation

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 .

1.1 Talking to the Interpreter

Simple Expressions

- Type an expression, and the interpreter evaulates 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
```

1.2 Evaluation and Termination

Evaluation

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

$$\begin{array}{ll} & \mathit{square}\ (3+4) \\ = & \{\ \text{definition of } +\ \} \\ & \mathit{square}\ 7 \\ = & \{\ \text{definition of } \mathit{square}\ \} \\ & 7 \times 7 \\ = & \{\ \text{definition of } \times\ \} \\ & 49 \end{array}$$

Another Evaluation Sequence

- Another possible reduction sequence:

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

- 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:

$$
\begin{aligned}
&\textbf{let}\ \mathit{three}\ x\ = 3 \\
&\textbf{let}\ \mathit{infinity}\ = \mathit{infinity} + 1
\end{aligned}
$$

  - Notice how the second definition is different from an assignment $\mathit{infinity} := \mathit{infinity} + 1$.
  - Note: The second definition is actually not accepted by OCaml. Instead one has to write

    $$\textbf{let rec}\ \mathit{infinity}\ () = \mathit{infinity}\ () + 1$$

    For clarity we temporarily use the current definition.

- Try evaluating *three infinity*. If we simplify *infinity* first:

$$
\begin{aligned}
& \mathit{three}\ \mathit{infinity} \\
= & \quad \{ \text{ definition of } \mathit{infinity} \ \} \\
& \mathit{three}\ (\mathit{infinity} + 1) \\
= & \ \mathit{three}\ ((\mathit{infinity} + 1) + 1) \ldots
\end{aligned}
$$

- If we start with simplifying *three*:

$$
\begin{aligned}
& \mathit{three}\ \mathit{infinity} \\
= & \quad \{ \text{ definition of } \mathit{three} \ \} \\
& 3
\end{aligned}
$$

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 $\bot$.
- A function f is *strict* if $f\ \bot = \bot$. Otherwise it is *non-strict*.
- Functions in OCaml are strict, thus OCaml is called a strict language.
- Haskell allows non-strict functions.

2 Functions

Mathematical Functions

- Mathematically, a function is a mapping between arguments and results.
  - A function $f : A \to B$ 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, $double = double'$:

  $$\begin{aligned} &\textbf{let}\ double\ x &= x + x \\ &\textbf{let}\ double'\ x &= 2 \times x \end{aligned}$$

  - 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.

2.1 Currying

Pairs

- Our first compound type: pairs.

  $\#\ \textbf{let}\ flolac12\ =\ (12, \texttt{"flolac"});;$
  $\textbf{val}\ flolac12\ :\ int * string\ =\ (12, \texttt{"flolac"})$

- **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*:

  $\#\ fst\ flolac12;;$
  $-\ :\ int\ =\ 12$
  $\#\ snd\ flolac12;;$
  $-\ :\ string\ =\ \texttt{"flolac"}$

Currying

- Consider again the function *smaller*:

  $\#\textbf{let}\ smaller\ (x, y)\ =\ \textbf{if}\ x \le y\ \textbf{then}\ x\ \textbf{else}\ y;;$
  $\textbf{val}\ smaller\ :\ 'a * 'a \to 'a\ =\ \langle fun \rangle$

- Another way:

  $\#\textbf{let}\ smallerc\ x\ y\ =\ \textbf{if}\ x \le y\ \textbf{then}\ x\ \textbf{else}\ y;;$
  $\textbf{val}\ smallerc\ :\ 'a \to 'a \to 'a\ =\ \langle fun \rangle$

  - It's a function returning a function.

    $\#smallerc\ 3;;$
    $- : int \to int = \langle fun \rangle$
    $\#smallerc\ 3\ 4;;$
    $- : int = 3$

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

Precedence and Association

- Syntax of some functional languages are talored toward encouraging the use of curried function.
- Type: $a \to b \to c = a \to (b \to c)$, not $(a \to b) \to c$.
- 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.

  $\#\textbf{let}\ \mathit{twice}\ f\ x\ =\ f\ (f\ x);;$
  $\textbf{val}\ \mathit{twice}\ :\ ('a \to 'a) \to 'a \to 'a = \langle \mathit{fun} \rangle$
  $\#\textbf{let}\ \mathit{quad}\ =\ \mathit{twice}\ \mathit{square};;$
  $\textbf{val}\ \mathit{quad}\ :\ \mathit{int} \to \mathit{int}\ =\ \langle \mathit{fun} \rangle$

- Try evaluating quad 3:

$$\begin{aligned} & \mathit{quad}\ 3 \\ = \ & \mathit{twice}\ \mathit{square}\ 3 \\ = \ & \mathit{square}\ (\mathit{square}\ 3) \\ = \ & \ldots \end{aligned}$$

- Had we defined:

  $\#\ \textbf{let}\ \mathit{twice}\ (f, x) = f\ (f\ x);;$
  $\textbf{val}\ \mathit{twice}\ :\ ('a \to 'a) * 'a \to 'a = \langle \mathit{fun} \rangle$

  we would have to write

  $\#\ \textbf{let}\mathit{quad}\ x = \mathit{twice}\ (\mathit{square}, x)$
  $\textbf{val}\ \mathit{quad}\ :\ \mathit{int} \to \mathit{int}\ =\ \langle \mathit{fun} \rangle$

Currying and Uncurrying

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

$\#\ \textbf{let}\ \mathit{curry}\ f\ x\ y\ =\ f\ (x, y)$
$\textbf{val}\ \mathit{curry}\ :\ ('a * 'b \to 'c) \to 'a \to 'b \to 'c\ =\ \langle \mathit{fun} \rangle$

  - We have $\mathit{curry}\ \mathit{smaller} = \mathit{smallerc}$.
  - We will explain the meaning of those types with "dots" later.

- Can you define $\mathit{uncurry}$?

  $\#\ \textbf{let}\ \mathit{uncurry} \ldots$
  $\textbf{val}\ \mathit{uncurry}\ :\ ('a \to 'b \to 'c) \to ('a * 'b \to 'c)$

2.2 Sectioning

Sectioning

- OCaml determines the fixty 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 $\mathit{int} \to \mathit{int} \to \mathit{int}$.

- Infix operator can be partially applied too.

  - $(+)\,1$ has type $\mathit{int} \to \mathit{int}$ and increments its argument by one.
  - $(/.)\,1$ has type $\mathit{float} \to \mathit{float}$. We have $(/.)\,1\,2 = 1/.2$.

Function Composition

- Functions composition:

  $\#\ \textbf{let}\ (\ll)\ f\ g\ x\ =\ f\ (g\ x);;$
  $\textbf{val}\ (\ll) : ('a \to 'b) \to ('c \to 'a) \to 'c \to 'b\ =\ \langle \mathit{fun} \rangle$

- Putting $(\ll)$ in infix position, we get $(f \ll g)\ x = f\ (g\ x)$.

- E.g. another way to write quad:

  $\textbf{let}\ \mathit{quad} = \mathit{square} \ll \mathit{square}$

- The identity function: $\textbf{let}\ \mathit{id}\ x = x$.

- Some important properties:

  - $\mathit{id} \ll f = f = f \ll \mathit{id}$.
  - $(f \ll g) \ll h = f \ll (g \ll h)$.

Anonymous Functions

- A function is a value that can be created, passed around, and used, like any other values.

- $\textbf{fun}\ x\ \to e$ is a function (without a name) that takes an argument x and yields e.

- Instead of $\textbf{let}\ \mathit{square}\ x\ =\ x * x$, one can also write

  $\textbf{let}\ \mathit{square}\ =\ \textbf{fun}\ x \to x * x$

- 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.

2.3 Definitions

Recursive Definitions

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

$$\begin{array}{l} \#\ \textbf{let rec}\ \mathit{fact}\ n\ =\\ \quad \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * \mathit{fact}\ (n-1);;\\ \textbf{val}\ \mathit{fact}\ :\ \mathit{int} \to \mathit{int}\ =\ \langle \mathit{fun}\rangle \end{array}$$

- How can something be defined in terms of itself? Are such definitions valid? We will talk about it later.

- Notice the difference:
  - **let rec** $f\ x\ =\ ...f...$ recursively defines f;
  - **let** $f\ x\ =\ ...f..$ 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:

$$\begin{array}{l} \#\ \textbf{let rec}\ \mathit{is_even}\ n\ =\\ \quad \textbf{if}\ n = 0\ \textbf{then}\ \mathit{true}\ \textbf{else}\ \mathit{is_odd}\ (n-1)\\ \textbf{and}\ \mathit{is_odd}\ n\ =\\ \quad \textbf{if}\ n = 0\ \textbf{then}\ \mathit{false}\ \textbf{else}\ \mathit{is_even}\ (n-1);;\\ \\ \textbf{val}\ \mathit{is_even}\ :\ \mathit{int} \to \mathit{bool}\\ \textbf{val}\ \mathit{is_odd}\ :\ \mathit{int} \to \mathit{bool} \end{array}$$

Pattern Matching

- The famous Fibonacci function:

$$\begin{array}{lcl} \textbf{let rec}\ \mathit{fib}\ n & = & \textbf{if}\ n = 0\ \textbf{then}\ 0\\ & & \textbf{else if}\ n = 1\ \textbf{then}\ 1\\ & & \textbf{else}\ \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2) \end{array}$$

- A equivalent definition using **match**:

$$\begin{array}{lcl} \textbf{let rec}\ \mathit{fib}\ n & = & \textbf{match}\ n\ \textbf{with}\\ & & \mid 0 \to 0\\ & & \mid 1 \to 1\\ & & \mid n \to \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2) \end{array}$$

- An abbreviation:

$$\begin{array}{lcl} \textbf{let rec}\ \mathit{fib} & = & \textbf{function}\\ & & \mid 0 \to 0\\ & & \mid 1 \to 1\\ & & \mid n \to \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2) \end{array}$$

Local Definitions

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

$$\begin{array}{lcl} \textbf{let}\ f\ (x, y) & = & \textbf{let}\ a = (x + y)/2\ \textbf{in}\\ & & \textbf{let}\ b = (x + y)/3\ \textbf{in}\\ & & (a + 1) * (b + 2) \end{array}$$

3 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 deducted 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.

$$\begin{array}{l} \#\ \textbf{let}\ f\ g\ x\ y\ =\ g\ (\mathit{char_of_int}\ x) + y;;\\ \textbf{val}\ f\ :\ (\mathit{char} \to \mathit{int}) \to \mathit{int} \to \mathit{int} \to \mathit{int}\ =\ \langle \mathit{fun}\rangle \end{array}$$

- 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

$$\begin{array}{l} \#\ \textbf{let}\ \textit{fork}\ f\ g\ x\ =\ (f\ x, g\ x) \\ \#\ \textit{fork}\ ((+)\ 1)\ ((\ *)\ 2)\ 5;; \\ -: \textit{int} * \textit{int} = (6, 10) \end{array}$$

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

Polymorphic Types

- Polymorphic types could be instantiated upon application.

$$\begin{array}{l} \#\ \textit{fork}\ \textit{int_of_char};; \\ -: (\textit{char} \to '_a) \to \textit{char} \to \textit{int} * '_a\ =\ \langle \textit{fun} \rangle \\ \#\ \textit{fork}\ \textit{char_of_int};; \\ -: (\textit{int} \to '_a) \to \textit{int} \to \textit{char} * '_a\ =\ \langle \textit{fun} \rangle \end{array}$$

Polymorphism

- Allowing values of different types to be handled through a uniform interface.
- Christopher Strachey descriped 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 ($Eq\ a \Rightarrow \ldots$) 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.

Part II
Induction on Datatypes

4 User-Defined Types

Variant, or Sum Types

- A simple example of a sum type:

$$\#\ \textbf{type}\ \textit{state}\ =\ \textit{On}\ \mid\ \textit{Off};;$$

  - The type *state* has two values

$$\begin{array}{l} \#\ \textit{On};; \\ -\ :\ \textit{state}\ =\ \textit{On} \\ \#\ \textit{Off};; \\ -\ :\ \textit{state}\ =\ \textit{Off} \end{array}$$

  - This type is isomorphic to *bool*. One can (and perhaps should) imagine that *bool* is actually defined this way. Alas, OCaml gives different syntax to primitive and user-defined types.
- Value of a sum type can be deconstructed by matching:

$$\begin{array}{l} \textbf{let}\ \textit{flip}\ =\ \textbf{function} \\ \qquad \mid\ \textit{On} \to \textit{Off} \\ \qquad \mid\ \textit{Off} \to \textit{On} \end{array}$$

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⟩
```

5 Induction on Natural Numbers

5.1 Datatypes, Functions, and Proofs

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 ℕ = 0 | 1+ ℕ
```

  - That is, any natural number is either 0, or **1+** n where n is a natural number.
  - Any natural number has the form: 0, **1+** 0, **1+** (**1+** 0) ... Decimal numbers 1, 2, 3, etc., can be seen as abbreviations.
  - The type ℕ is the smallest set such that
    1. 0 is in ℕ;
    2. if n is in ℕ, so is **1+** n.
  - Thus to show that P holds for all natural numbers, we only need to consider these two cases.

Inductively Defined Functions

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

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

[1] Not a real OCaml definition.

- Even addition can be defined inductively

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

- Exercise: define $(*)$?

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

Case $m := 0$:

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

Proof by Induction

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

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

Structure Proofs by Programs

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

Lists and Natural Numbers

- We have yet to prove that (×) is associative.
- The proof is quite similar to the proof for associativity of (@), which we will talk about later.
- In fact, 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 ℕ as given.

5.2 What is a Proof, Anyway?

But What is a Proof, Anyway?

Xavier Leroy, "How to prove it" `http://cristal.inria.fr/~xleroy/stuff/how-to-prove-it.html`:

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).'

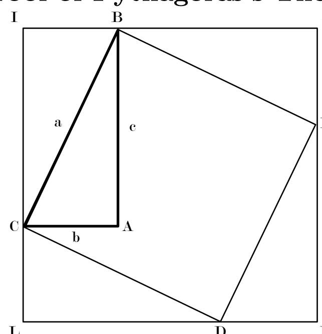
Proof by appeal to intuition Cloud-shaped drawings.

A semantic proof

A map of London is place 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. [**?**, Figure 3.2]

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 $\widehat{BAC} = 90^o$. 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

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

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.

- 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.
- 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.
- Junior high school: algebra!

$$\begin{aligned} x + y &= 5 \\ 2x + 4y &= 14. \end{aligned}$$

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

Formal Proof
Recall our proof

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

It has a rather formal taste.

6 Induction on Lists

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 $(int \to int \to int)\ list$.
- $[\,]$, 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 $(::) :: {'a} \to {'a}\ list \to {'a}\ list$ 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

$$\textbf{type}\ {'a}\ list\ =\ [\,] \mid {'a} :: {'a}\ list$$

- $[1;2;3]$ is an abbreviation of $1 :: (2 :: (3 :: [\,]))$.

Head and Tail

- The following functions are in module *List*. You have to issue a command
- $hd : 'a\ list \to 'a$. e.g. $hd\ [1;2;3] = 1$.
- $tl : 'a\ list \to 'a\ list$. 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).
- $length : 'a\ list \to int$ returns the length of a list.

Finite Lists are Inducitvely Defined

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

$$\textbf{type}\ 'a\ list\ =\ [\,]\ |\ 'a :: 'a\ list$$

- 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.

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⟩
```

  - $sum\ [1;2;3;4;5] = 15$

[2]Not a real OCaml definition.

- 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
```

  - $map\ ((+)\ 1)\ [1;2;3;4] = [2;3;4;5]$
  - $map\ (\textbf{fun}\ x \to x * 2)\ [1;2;3;4] = [2;4;6;8]$

6.1 Append, and Some of Its Properties

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 $(+)$!

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 $xs := [\,]$:

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

[3]This function has an alias: *append*.

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:

$$
\begin{aligned}
& \textbf{let rec } \mathit{length} \ = \ \textbf{function} \\
& \quad \mid \ [\,] \to 0 \\
& \quad \mid \ x :: xs \to 1 + \mathit{length} \ xs \\
& \textbf{val } \mathit{length} : {}'a \ \mathit{list} \to \mathit{int}
\end{aligned}
$$

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

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

Concatenation

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

$$
\begin{aligned}
& \textbf{let rec } \mathit{concat} \ = \ \textbf{function} \\
& \quad \mid [\,] \to [\,] \\
& \quad \mid xs :: xss \to xs @ \mathit{concat} \ xss \\
& \textbf{val } \mathit{concat} : {}'a \ \mathit{list} \ \mathit{list} \to {}'a \ \mathit{list}
\end{aligned}
$$

- Compare with *sum*.
- Exercise: prove $\mathit{sum} \ll \mathit{concat} = \mathit{sum} \ll \mathit{map} \ \mathit{sum}$.

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.

6.2 More Inductively Defined Functions

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.)
- **Note** Terminology: an inductive definition, as we have seen, define "bigger" things in terms of "smaller" things. Recursion, on the other hand, is a more general term, meaning "to define one entity in terms of itself."
- 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*.

$$
\begin{aligned}
& \textbf{let rec } \mathit{filter} \ p \ = \ \textbf{function} \\
& \quad \mid [\,] \to [\,] \\
& \quad \mid x :: xs \to \textbf{if } p \, x \textbf{ then } x :: \mathit{filter} \ p \ xs \\
& \qquad\qquad\qquad\qquad \textbf{else } \mathit{filter} \ p \ xs \\
& \textbf{val } \mathit{filter} \ : \ ({}'a \to \mathit{bool}) \to {}'a \ \mathit{list} \to {}'a \ \mathit{list}
\end{aligned}
$$

Take and Drop

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

$$
\begin{aligned}
& \textbf{let rec } \mathit{take} \ n \ xs = \textbf{match } n, xs \textbf{ with} \\
& \quad \mid 0, _ \to [\,] \\
& \quad \mid (\mathbf{1}\!+ n), \ [\,] \to [\,] \\
& \quad \mid (\mathbf{1}\!+ n), \ (x :: xs) \to x :: \mathit{take} \ n \ xs \\
& \textbf{val } \mathit{take} \ : \ \mathit{int} \to {}'a \ \mathit{list} \to {}'a \ \mathit{list}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let rec } \mathit{drop}\ n\ \mathit{xs} = \textbf{match } n, \mathit{xs} \textbf{ with}\\
&\qquad |\ 0, \mathit{xs} \to \mathit{xs}\\
&\qquad |\ (\mathbf{1}{+}\ n),\ [\,] \to [\,]\\
&\qquad |\ (\mathbf{1}{+}\ n),\ (x :: \mathit{xs}) \to \mathit{drop}\ n\ \mathit{xs}\\
&\textbf{val } \mathit{drop}\ :\ \mathit{int} \to {}'a\ \mathit{list} \to {}'a\ \mathit{list}
\end{aligned}
$$

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

TakeWhile and DropWhile

- $\mathit{takeWhile}\ p\ \mathit{xs}$ yields the longest prefix of xs such that p holds for each element.

$$
\begin{aligned}
&\textbf{let rec } \mathit{takeWhile}\ p\ =\ \textbf{function}\\
&\qquad |\ [\,] \to [\,]\\
&\qquad |\ x :: \mathit{xs} \to \textbf{if } p\ x \textbf{ then } x :: \mathit{takeWhile}\ p\ \mathit{xs}\\
&\qquad\qquad\qquad\qquad \textbf{else } [\,];;\\
&\textbf{val } \mathit{takeWhile}\ :\ ({}'a \to \mathit{bool}) \to {}'a\ \mathit{list} \to {}'a\ \mathit{list}
\end{aligned}
$$

- $\mathit{dropWhile}\ p\ \mathit{xs}$ drops the prefix from xs.

$$
\begin{aligned}
&\textbf{let rec } \mathit{dropWhile}\ p\ =\ \textbf{function}\\
&\qquad |\ [\,] \to [\,]\\
&\qquad |\ x :: \mathit{xs} \to \textbf{if } p\ x \textbf{ then } \mathit{dropWhile}\ p\ \mathit{xs}\\
&\qquad\qquad\qquad\qquad \textbf{else } x :: \mathit{xs};;\\
&\textbf{val } \mathit{dropWhile}\ :\ ({}'a \to \mathit{bool}) \to {}'a\ \mathit{list} \to {}'a\ \mathit{list}
\end{aligned}
$$

- Prove: $\mathit{takeWhile}\ p\ \mathit{xs} @ \mathit{dropWhile}\ p\ \mathit{xs} = \mathit{xs}$.

List Reversal

- $\mathit{reverse}\ [1;2;3;4] = [4;3;2;1]$.

$$
\begin{aligned}
&\textbf{let rec } \mathit{reverse}\ =\ \textbf{function}\\
&\qquad |\ [\,] \to [\,]\\
&\qquad |\ x :: \mathit{xs} \to \mathit{reverse}\ \mathit{xs} @ [x];;\\
&\textbf{val } \mathit{reverse}\ :\ {}'a\ \mathit{list} \to {}'a\ \mathit{list}
\end{aligned}
$$

All Prefixes and Suffixes

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

Totality

- Structure of our definitions so far:

$$
\begin{aligned}
&\textbf{let rec } f\ =\ \textbf{function}\\
&\qquad |\ \ [\,] \to \ldots\\
&\qquad |\ \ x :: \mathit{xs} \to \ldots f\ \mathit{xs} \ldots
\end{aligned}
$$

  - 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, guranteeing termination.
- Together they guarantee that every input is mapped to some output. Thus they define *total* functions on lists.

6.3 Other Patterns of Induction

Variations with the Base Case

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

$$
\begin{aligned}
&\textbf{let rec } \mathit{fib}\ =\ \textbf{function}\\
&\qquad |\ 0 \to 0\\
&\qquad |\ 1 \to 1\\
&\qquad |\ n \to \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)
\end{aligned}
$$

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

$$
\begin{aligned}
&\textbf{let rec } f\ =\ \textbf{function}\\
&\qquad |\ \ [x] \to \ldots x \ldots\\
&\qquad |\ \ x :: \mathit{xs} \to \ldots f\ \mathit{xs} \ldots
\end{aligned}
$$

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

$$\textbf{type } {}'a\ \mathit{list}^{+}\ =\ [a]\ |\ a :: {}'a\ \mathit{list}^{+}$$

  - We do not want to define *map*, *filter* again for ${}'a\ \mathit{list}^{+}$. Thus we reuse ${}'a\ \mathit{list}$ and pretend that we were talking about ${}'a\ \mathit{list}^{+}$.
  - 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:

$$
\begin{array}{l}
\textbf{let rec } \mathit{merge}\ \mathit{xs}\ \mathit{ys} \ = \ \textbf{match } \mathit{xs}, \mathit{ys} \textbf{ with} \\
\quad \mid [\,], [\,] \qquad \to [\,] \\
\quad \mid [\,], y :: \mathit{ys} \quad \to y :: \mathit{ys} \\
\quad \mid x :: \mathit{xs}, [\,] \quad \to x :: \mathit{xs} \\
\quad \mid x :: \mathit{xs}, y :: \mathit{ys} \to \textbf{if } x \leq y \\
\qquad\qquad\qquad\qquad \textbf{then } x :: \mathit{merge}\ \mathit{xs}\ (y :: \mathit{ys}) \\
\qquad\qquad\qquad\qquad \textbf{else } y :: \mathit{merge}\ (x :: \mathit{xs})\ \mathit{ys}
\end{array}
$$

Non-Structural Induction

- In most of the programs we've seen so far, the recursive call 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 implemenation of mergesort below, for example, the arguments always get smaller in size.

$$
\begin{array}{l}
\textbf{let rec } \mathit{msort} \ = \ \textbf{function} \\
\quad [\,] \ \to [\,] \\
\quad [x] \to [x] \\
\quad \mathit{xs} \ \to \textbf{let } n = \mathit{length}\ \mathit{xs}\ /\ 2 \textbf{ in} \\
\qquad\quad \textbf{let } \mathit{ys} = \mathit{take}\ n\ \mathit{xs} \textbf{ in} \\
\qquad\quad \textbf{let } \mathit{zs} = \mathit{drop}\ n\ \mathit{xs} \textbf{ in} \\
\qquad\quad \mathit{merge}\ (\mathit{msort}\ \mathit{ys})\ (\mathit{msort}\ \mathit{zs})
\end{array}
$$

  - 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:

$$
\begin{array}{l}
\textbf{let rec } f \ = \ \textbf{function} \\
\quad \mid 0 \to 0 \\
\quad \mid n \to f\, n
\end{array}
$$

- 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

7 On Efficiency of Operations on Lists

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.

$$
\begin{array}{l}
\textbf{let rec } \mathit{init} \ = \ \textbf{function} \\
\quad \mid [x] \to [\,] \\
\quad \mid x :: \mathit{xs} \to x :: \mathit{init}\ \mathit{xs}
\end{array}
$$

- Consider *init* $[1;2;3;4]$:

$$\begin{aligned}
& \mathit{init}\ (1 :: 2 :: 3 :: 4 :: [\,]) \\
= {} & 1 :: \mathit{init}\ (2 :: 3 :: 4 :: [\,]) \\
= {} & 1 :: 2 :: \mathit{init}\ (3 :: 4 :: [\,]) \\
= {} & 1 :: 2 :: 3 :: \mathit{init}\ (4 :: [\,]) \\
= {} & 1 :: 2 :: 3 :: [\,]
\end{aligned}$$

List Concatenation Takes Linear Time

- Recall (@):

$$\begin{aligned}
& \textbf{let rec}\ (@)\ xs\ ys = \textbf{match}\ xs\ \textbf{with} \\
& \quad \mid [\,] \to ys \\
& \quad \mid x :: xs \to x :: (xs\ @\ ys) \\
& \textbf{val}\ (@)\ :\ 'a\ list \to 'a\ list \to 'a\ list
\end{aligned}$$

- Consider $[1;2;3]\ @[4;5]$:

$$\begin{aligned}
& (1 :: 2 :: 3 :: [\,])\ @(4 :: 5 :: [\,]) \\
= {} & 1 :: ((2 :: 3 :: [\,])\ @(4 :: 5 :: [\,])) \\
= {} & 1 :: 2 :: ((3 :: [\,])\ @(4 :: 5 :: [\,])) \\
= {} & 1 :: 2 :: 3 :: ([\,]\ @(4 :: 5 :: [\,])) \\
= {} & 1 :: 2 :: 3 :: 4 :: 5 :: [\,]
\end{aligned}$$

- (@) 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 $O(n)$, where n is the length of the list.
- 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:

$$\begin{aligned}
& \textbf{let rec}\ \mathit{reverse}\ =\ \textbf{function} \\
& \quad \mid [\,] \to [\,] \\
& \quad \mid x :: xs \to \mathit{reverse}\ xs\ @[x];; \\
& \textbf{val}\ \mathit{reverse}\ :\ 'a\ list \to 'a\ list
\end{aligned}$$

- E.g. $\mathit{reverse}\ [1;2;3;4] = ((([\,]\ @[4])\ @[3])\ @[2])\ @[1] = [4;3;2;1]$.
- 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.

8 Batched Queue

Persistency

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

$$\begin{aligned}
& \textbf{let}\ x = [1;2;3;4]\ \textbf{in} \\
& \textbf{let}\ x' = \mathit{reverse}\ x\ \textbf{in} \\
& \textbf{let}\ x'' = \mathit{tl}\ x' \ldots
\end{aligned}$$

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:

$$\begin{aligned}
& \textbf{let}\ x = [1;2;3;4]\ \textbf{in} \\
& \textbf{let}\ x' = \mathit{set}\ x\ 0\ 2\ \textbf{in} \\
& \textbf{let}\ x'' = \mathit{set}\ x\ 0\ 3 \ldots
\end{aligned}$$

  - 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:

$$\begin{aligned}
& \textbf{type}\ 'a\ queue\ =\ 'a\ list \\
& \textbf{let}\ \mathit{head}\ xs\ =\ \mathit{hd}\ xs \\
& \textbf{let}\ \mathit{tail}\ xs\ =\ \mathit{tl}\ xs \\
& \textbf{let}\ \mathit{snoc}\ xs\ y\ =\ xs\ @[y]
\end{aligned}$$

- 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.

$$
\begin{array}{ll}
 & ([1;2],[5;4;3]) \\
\longrightarrow & \quad \{\ \text{remove } 1\ \} \\
 & ([2],[5;4;3]) \\
\longrightarrow & \quad \{\ \text{add } 6\ \} \\
 & ([2],[6;5;4;3]) \\
\longrightarrow & \quad \{\ \text{remove } 2\ \} \\
 & ([\,],[6;5;4;3]) \\
\longrightarrow & \quad \{\ \text{shifting elements}\ \} \\
 & ([3;4;5;6],[\,]) \\
\longrightarrow & \quad \{\ \text{remove } 3\ \} \\
 & ([4;5;6],[\,])
\end{array}
$$

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.

FILO Queue: Methods

- Define **type** $'a\ queue = 'a\ list * 'a\ list$.
- Goal: define the following methods:
  - $empty : 'a\ queue$
  - $is_empty : 'a\ queue \to bool$
  - $head : 'a\ queue \to 'a$
  - $tail : 'a\ queue \to 'a\ queue$
  - $snoc : 'a\ queue \to 'a \to 'a\ queue$

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 → x

let tail = function
  | x :: xs, ys → shift (xs, ys)
```

Shifting and Adding Elements

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

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

9 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:

$$\begin{aligned}&\textbf{let rec } \mathit{height} \ = \ \textbf{function}\\ &\quad |\ \mathit{Empty} \to 0\\ &\quad |\ \mathit{Node}\ (t, _, u) \to\\ &\qquad 1 + \mathit{max}\ (\mathit{height}\ t)\ (\mathit{height}\ u)\end{aligned}$$

- Size of a tree:

$$\begin{aligned}&\textbf{let rec } \mathit{size} \ = \ \textbf{function}\\ &\quad |\ \mathit{Empty} \to 0\\ &\quad |\ \mathit{Node}\ (t, _, u) \to 1 + \mathit{size}\ t + \mathit{size}\ u\end{aligned}$$

Inductively Defined Functions on Trees

- Elements of a tree:

$$\begin{aligned}&\textbf{let rec } \mathit{flatten} \ = \ \textbf{function}\\ &\quad |\ \mathit{Empty} \to [\,]\\ &\quad |\ \mathit{Node}\ (t, x, u) \to\\ &\qquad \mathit{flatten}\ t\ @ [x]\ @\ \mathit{flatten}\ u\end{aligned}$$

- 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 $\mathit{Node}\ (t, x, u)$, all elements in t are smaller than x, and all elements in u are greater than x.
- Define the following methods:
  - $\mathit{member} : {}'a \to {}'a\ \mathit{tree} \to \mathit{bool}$
  - $\mathit{insert} : {}'a \to {}'a\ \mathit{tree} \to {}'a\ \mathit{tree}$

Membership

$$\begin{aligned}&\textbf{let rec } \mathit{member}\ x \ = \ \textbf{function}\\ &\quad |\ \mathit{Empty} \to \mathit{false}\\ &\quad |\ \mathit{Node}\ (t, y, u) \to\\ &\qquad \textbf{if } x < y \textbf{ then } \mathit{member}\ x\ t\\ &\qquad \textbf{else if } y < x \textbf{ then } \mathit{member}\ x\ u\\ &\qquad \textbf{else } \mathit{true}\end{aligned}$$

Insertion

- Inserting an element into a tree:

$$\begin{aligned}&\textbf{let rec } \mathit{insert}\ x \ = \ \textbf{function}\\ &\quad |\ \mathit{Empty} \to \mathit{Node}\ (\mathit{Empty}, x, \mathit{Empty})\\ &\quad |\ \mathit{Node}\ (t, y, u) \textbf{ as } s \to\\ &\qquad \textbf{if } x < y \textbf{ then } \mathit{Node}\ (\mathit{insert}\ x\ t, y, u)\\ &\qquad \textbf{else if } y > x \textbf{ then } \mathit{Node}\ (t, y, \mathit{insert}\ x\ u)\\ &\qquad \textbf{else } s\end{aligned}$$

- 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

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." [**?**]

"The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand." [**?**]

- 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

$$
\begin{array}{ll}
 & \mathit{max}\ \{\ \mathit{sum}\ (i,j) \mid 0 \le i \le j \le N\ \} \\
= & \{\ \text{Premise 1}\ \} \\
 & \mathit{max} \ll \mathit{map}\ \mathit{sum} \ll \mathit{concat} \ll \mathit{map}\ \mathit{inits} \ll \mathit{tails} \\
= & \{\ \text{Premise 2}\ \} \\
 & \ldots \\
= & \{\ \ldots\ \} \\
 & \text{The final program!}
\end{array}
$$

10 The Unfold/Fold Transformation

Sum and Map

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

$$
\begin{array}{l}
\#\ \textbf{let rec}\ \mathit{sum}\ =\ \textbf{function} \\
\quad \mid [\,] \to 0 \\
\quad \mid x :: \mathit{xs} \to x + \mathit{sum}\ \mathit{xs};;
\end{array}
$$

  - E.g. $\mathit{sum}\ [7;9;11] = 27$.

- The function *map f* takes a list and builds a new list by applying f to every item in the input:

$$
\begin{array}{l}
\#\ \textbf{let rec}\ \mathit{map}\ f\ =\ \textbf{function} \\
\quad \mid [\,] \to [\,] \\
\quad \mid x :: \mathit{xs} \to f\ x :: \mathit{map}\ f\ \mathit{xs};; \\
\textbf{val}\ \mathit{map} : ('a \to 'b) \to 'a\ \mathit{list} \to 'b\ \mathit{list}
\end{array}
$$

  - E.g. $\mathit{map}\ \mathit{square}\ [3;4;6] = [9;16;36]$.

10.1 Example: Sum of Squares

Sum of Squares

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

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

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

An Alternative Definition for *sumsq*

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

$$\begin{aligned}&\textbf{let}\ sumsq\ =\ \textbf{function}\\ &\quad \mid [\,] \to 0\\ &\quad \mid x :: xs \to square\ x + sumsq\ xs\end{aligned}$$

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

$$\begin{aligned}f\ [\,] &= 0\\ f\ (x :: xs) &= square\ x + f\ xs\end{aligned}$$

- 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.

11 Accumulating Parameters

Reversing a List

- The function *reverse* was defined by:

$$\begin{aligned}&\textbf{let rec}\ reverse\ =\ \textbf{function}\\ &\qquad \mid [\,] \to [\,]\\ &\qquad \mid x :: xs \to reverse\ xs @ [x];;\\ &\textbf{val}\ reverse\ :\ 'a\ list \to 'a\ list\end{aligned}$$

- 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?

11.1 Fast List Reversal

Introducing an Accumulating Parameter

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

$$\begin{aligned}&\textbf{let}\ revcat\ xs\ ys\ =\ reverse\ xs @ ys\\ &\textbf{val}\ revcat\ :\ 'a\ list \to 'a\ list \to 'a\ list\end{aligned}$$

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

$$\textbf{let}\ reverse\ xs\ =\ revcat\ xs\ [\,]$$

Reversing a List, Base Case

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

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

Reversing a List, Inductive Case

Case $x :: xs$:

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

Linear-Time List Reversal

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

$$\begin{aligned}&\textbf{let rec}\ revcat\ xs\ ys\ =\ \textbf{match}\ xs\ \textbf{with}\\ &\quad \mid\ [\,] \to ys\\ &\quad \mid\ x :: xs \to revcat\ xs\ (x :: ys)\end{aligned}$$

- 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?

11.2 Tail Recursion and Loops

Tracing Reverse

$$
\begin{array}{ll}
 & reverse\ [1;2;3;4] \\
= & revcat\ [1;2;3;4]\ [\,] \\
= & revcat\ [2;3,4]\ [1] \\
= & revcat\ [3;4]\ [2;1] \\
= & revcat\ [4]\ [3;2;1] \\
= & revcat\ [\,]\ [4;3;2;1] \\
= & [4;3;2;1]
\end{array}
$$

$$
\begin{array}{l}
\mathbf{let}\ reverse\ xs\ =\ revcat\ xs\ [\,] \\
\mathbf{let\ rec}\ revcat\ xs\ ys\ =\ \mathbf{match}\ xs\ \mathbf{with} \\
\quad \mid\ [\,] \to ys \\
\quad \mid\ x :: xs \to revcat\ xs\ (x :: ys)
\end{array}
$$

$$
\begin{array}{l}
xs, ys\ \leftarrow XS, [\,]; \\
\mathbf{while}\ xs \neq [\,]\ \mathbf{do} \\
\qquad xs, ys\ \leftarrow\ (tl\ xs), (hd\ xs :: ys); \\
\mathbf{return}\ ys
\end{array}
$$

Tail Recursion

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

$$
\begin{array}{l}
f\ x_1\ \ldots\ x_n = \{\text{base case}\} \\
f\ x_1\ \ldots\ x_n = f\ x'_1\ \ldots\ x'_n
\end{array}
$$

- 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:

$$
\begin{array}{l}
\mathbf{let}\ sumsq\ =\ \mathbf{function} \\
\quad \mid [\,] \to 0 \\
\quad \mid x :: xs \to square\ x + sumsq\ xs
\end{array}
$$

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

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

11.3 Being Quicker by Doing More!

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 difficulty 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:

$$
\begin{array}{l}
\textbf{let rec } \mathit{combine}\ \mathit{xs}\ \mathit{ys}\ =\ \textbf{match } \mathit{xs}, \mathit{ys}\ \textbf{with} \\
\quad \mid\ [\,], [\,] \to [\,] \\
\quad \mid\ x :: \mathit{xs}, y :: \mathit{ys} \to (x, y) :: \mathit{combine}\ \mathit{xs}\ \mathit{ys} \\
\textbf{val } \mathit{combine}\ :\ '\mathit{a}\ \mathit{list} \to '\mathit{b}\ \mathit{list} \to ('\mathit{a} * '\mathit{b})\ \mathit{list}
\end{array}
$$

- E.g. $\mathit{combine}\ [1;3;5]\ [\texttt{'}a\texttt{'};\texttt{'}b\texttt{'};\texttt{'}c\texttt{'}] = [(1,\texttt{'}a\texttt{'});(3,\texttt{'}b\texttt{'});(5;\texttt{'}c\texttt{'})]$.

Range Generation

- Generating a range:

$$
\begin{array}{l}
\textbf{let } \mathit{fromTo}\ m\ n\ =\ \textbf{if } m \geq n\ \textbf{then } [\,] \\
\qquad\qquad\qquad\qquad \textbf{else } m :: \mathit{fromTo}\ (m+1)\ n
\end{array}
$$

- E.g. $\mathit{fromTo}\ 3\ 6\ =\ [3;4;5]$.
- E.g. $\mathit{fromTo}\ 6\ 6\ =\ [\,]$.

Labelling a List

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

$$
\begin{array}{l}
\textbf{let } \mathit{index}\ \mathit{xs} = \mathit{combine}\ (\mathit{fromTo}\ 0\ (\mathit{length}\ \mathit{xs}))\ \mathit{xs} \\
\textbf{val } \mathit{index}\ :\ '\mathit{a}\ \mathit{list} \to (\mathit{int} * '\mathit{a})\ \mathit{list}
\end{array}
$$

- E.g. $\mathit{index}\ [\texttt{'}a\texttt{'};\texttt{'}b\texttt{'};\texttt{'}c\texttt{'}] = [(0,\texttt{'}a\texttt{'});(1,\texttt{'}b\texttt{'});(2;\texttt{'}c\texttt{'})]$.

Labelling a List, Inductive Case

-

- To construct an inductive definition, the case for $[\,]$ is easy. For the $x :: \mathit{xs}$ case:

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

- 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*:

$$
\begin{array}{l}
\textbf{let } \mathit{idxFrom}\ \mathit{xs}\ n = \\
\qquad \mathit{combine}\ (\mathit{fromTo}\ n\ (n + \mathit{length}\ \mathit{xs}))\ \mathit{xs}
\end{array}
$$

- Initialisation: $\textbf{let } \mathit{index}\ \mathit{xs} = \mathit{idxFrom}\ \mathit{xs}\ 0$.
- We reason:

$$
\begin{array}{ll}
 & \mathit{idxFrom}\ (x :: \mathit{xs})\ n \\
= & \mathit{combine}\ (\mathit{fromTo}\ n\ (n + \mathit{len}\ (x :: \mathit{xs})))\ (x :: \mathit{xs}) \\
= & \mathit{combine}\ (\mathit{fromTo}\ n\ (1 + n + \mathit{len}\ \mathit{xs}))\ (x :: \mathit{xs}) \\
= & \mathit{combine}\ (n :: \mathit{fromTo}\ (1+n)\ (1 + n + \mathit{len}\ \mathit{xs}))\ (x :: \mathit{xs}) \\
= & (n, x) : \mathit{combine}\ (\mathit{fromTo}\ (1+n)\ (\mathit{len}\ \mathit{xs}))\ \mathit{xs} \\
= & (n, x) : \mathit{idxFrom}\ \mathit{xs}\ (1+n)
\end{array}
$$

Labelling a List, Second Attempt

$$
\begin{array}{l}
\textbf{let } \mathit{index}\ \mathit{xs} = \mathit{idxFrom}\ \mathit{xs}\ 0 \\
\\
\textbf{let } \mathit{idxFrom}\ \mathit{xs}\ n\ =\ \textbf{match } \mathit{xs}\ \textbf{with} \\
\quad \mid [\,] \to [\,] \\
\quad \mid x :: \mathit{xs} \to (n, x) : \mathit{idxFrom}\ \mathit{xs}\ (1+n)
\end{array}
$$

11.4 Proof by Strengthening

Summing Up a List in Reverse

- Prove: $\mathit{sum} \ll \mathit{reverse} = \mathit{sum}$, using the definition $\mathit{reverse}\ \mathit{xs} = \mathit{revcat}\ \mathit{xs}\ [\,]$. That is, proving $\mathit{sum}\ (\mathit{revcat}\ \mathit{xs}\ [\,]) = \mathit{sum}\ \mathit{xs}$.
- Base case trivial. For the case $x :: \mathit{xs}$:

$$
\begin{array}{ll}
 & \mathit{sum}\ (\mathit{reverse}\ (x :: \mathit{xs})) \\
= & \mathit{sum}\ (\mathit{revcat}\ (x :: \mathit{xs})\ [\,]) \\
= & \mathit{sum}\ (\mathit{revcat}\ \mathit{xs}\ [x])
\end{array}
$$

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

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

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

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

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:

$$\begin{array}{l} (\forall n : \ldots : take\, n\, x \leq_d drop\, n\, x) \\ \quad \Rightarrow\ maximum\, (map\, ((@) z)\, (inits\, x)) \\ \qquad = z \uparrow_d (z\, @\, x) \end{array}$$

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

$$\begin{array}{l} (\forall n : \ldots : y\, @\, take\, n\, x \leq_d drop\, n\, x) \\ \quad \Rightarrow\ z \uparrow_d maximum\, (map\, ((@)(z\, @\, y))\, (inits\, x)) \\ \qquad = z \uparrow_d (z\, @\, y\, @\, x) \end{array}$$

- 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.

12 Tupling

Steep Lists

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

$$\begin{array}{l} \textbf{let rec}\ steep\ =\ \textbf{function} \\ \quad \mid [\,] \rightarrow true \\ \quad \mid x :: xs \rightarrow steep\ xs\ \ \&\&\ \ x > sum\ xs \\ \textbf{val}\ steep\ :\ int\ list \rightarrow bool \end{array}$$

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

$$\textbf{let}\ steepsum\ xs\ =\ (steep\ xs, sum\ xs)$$

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

$$steepsum\ [\,] = (true, 0).$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

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

Synthesised Program

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

$$
\begin{array}{l}
\textbf{let}\ \mathit{steep}\ =\ \mathit{fst} \ll \mathit{steepsum} \\
\textbf{let}\ \mathit{steepsum}\ =\ \textbf{function} \\
\quad \mid [\,] \rightarrow (\mathit{true}, 0) \\
\quad \mid x :: xs \rightarrow \textbf{match}\ \mathit{steepsum}\ xs\ \textbf{with} \\
\qquad\qquad\qquad \mid b, y \rightarrow (b\ \&\&\ x > y, x + y)
\end{array}
$$

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

How Far Can We Go?

- We will show in Appendix **??** the entire derivation of the maximum segment sum problem.

- Bird and de Moor [**?**] conducted a though 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.

References

[Bac03] Roland C. Backhouse. *Program Construction: Calculating Implementations from Specifications.* John Wiley & Sons, Ltd., 2003.

[BdM97] Richard S. Bird and Oege de Moor. *Algebra of Programming.* International Series in Computer Science. Prentice Hall, 1997.

[Bir98] Richard S. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[CM98] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998.

[CMP05] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications with Objective Caml.* O'Reilly, 2005.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. EWD 340, Turing Award lecture.

[Dij74] Edsger W. Dijkstra. Programming as a discipline of mathematical nature. *American Mathematical Monthly*, 81(6):608–612, May 1974. EWD 361.

[Hut07] Graham Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

[Oka99] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1999.

[OSG98] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell.* O'Reilly, 1998.

A Fixty, Associativity, and Precedences

In OCaml, whether an operator is infix or prefix, infix operator associativity (left or right) and operator precedences (which of + and $*$ is stronger?) are syntactically determined by the first character of the operator.

! ~ ?	prefix
= < > | & $	infix0, left
@ ∧	infix1, right
+ −	infix2, left
∗ /	infix3, left

The operator ∗∗ is exceptional. It is right associative and has power 4.

B Maximum Segment Sum

In this appendix we will denote ≪ by $(\cdot)$, as is more common in this community. We also abbreviate **fun** $xs \rightarrow x :: xs$ to $(x ::)$.[4] To save space, the *maximum* of OCaml is written *max*, while OCaml's *max* is written ↑ infixed.

The maximum segment sum problem (*mss*) can be specified by

$$\textbf{let}\ mss = max \cdot map\ sum \cdot segments,$$

where $segments = concat \cdot map\ inits \cdot tails$. That is, the specification enumerates all segments of the input list, computes the sum of each of the segment, and pick the maximum. Also recall the definitions of *inits* and *tails* (not *init* and *tail*!):

$$\begin{array}{l}\textbf{let rec}\ inits\ =\ \textbf{function}\\ \quad \mid [\,] \rightarrow [[\,]]\\ \quad \mid x :: xs \rightarrow [\,] :: map\ (x ::)\ (inits\ xs),\end{array}$$

$$\begin{array}{l}\textbf{let rec}\ tails\ =\ \textbf{function}\\ \quad \mid [\,] \rightarrow [[\,]]\\ \quad \mid x :: xs \rightarrow (x :: xs) :: tails\ xs,\end{array}$$

and let *max* be defined on non-empty lists:

$$\begin{array}{l}\textbf{let rec}\ max\ =\ \textbf{function}\\ \quad \mid [x] \rightarrow x\\ \quad \mid x :: xs \rightarrow x \uparrow max\ xs,\end{array}$$

where ↑ returns the larger of its two arguments.

[4] Thus makes the notation more like Haskell.

We start with considering a simpler problem: given a list, compute the maximum sum among its prefixes. Denote this problem by *mps* (maximum prefix sum):

$$mps\ = max \cdot map\ sum \cdot inits.$$

Can we come up with an inductive definition of *mps*? Yes, you can already do that using what you have learned. The base case for $[\,]$ is easy. For the inductive case:

$$\begin{array}{ll} & mps\ (x :: xs)\\ = & max\ (map\ sum\ (inits\ (x :: xs)))\\ = & max\ \ (map\ sum\ ([\,] :: map\ (x ::)\ (inits\ xs)))\\ = & max\ \ (0 :: map\ sum\ (map\ (x ::)\ (inits\ xs)))\\ = & \quad \{\ \ map\ f \cdot map\ g = map\ (f \cdot g)\ \ \}\\ & max\ (0 :: map\ (sum \cdot (x ::))\ (inits\ xs))\\ = & \quad \{\ \ sum\ (x : ys) = x + sum\ ys\ \ \}\\ & max\ (0 :: map\ ((x+) \cdot sum)\ (inits\ xs))\\ = & \quad \{\ \ \text{defn. of}\ max\ \ \}\\ & 0 \uparrow max\ (map\ ((x+) \cdot sum)\ (inits\ xs))\\ = & \quad \{\ \ max\ (map\ (x+)\ ys) = x + max\ ys\ \ \}\\ & 0 \uparrow (x + max\ (map\ sum\ (inits\ xs)))\\ = & 0 \uparrow (x + mps\ xs)\end{array}$$

Thus we have an inductive definition for *mps*:

$$\begin{array}{l}\textbf{let rec}\ mps\ =\ \textbf{function}\\ \quad \mid [\,] \rightarrow 0\\ \quad \mid x :: xs \rightarrow 0 \uparrow (x + mps\ xs),\end{array}$$

which runs in linear time. The key step is the one using the lemma that $max\ (map\ (x+)\ ys) = x + max\ ys$. It needs a separate proof using the fact:

$$(x + y) \uparrow (x + z) = x + (y \uparrow z),$$

that is, addition distributes over maximum. This is the key property that makes an efficient implementation of *mps* (and thus *mss*) possible.

How is *mps* related to *mss*? In fact, solutions of many segment problems start with factoring the problem into the form computing "optimal prefix for each suffix". Here is how it works for *mss*:

$$\begin{array}{ll} & max \cdot map\ sum \cdot segments\\ = & max \cdot map\ sum \cdot concat \cdot map\ inits \cdot tails\\ = & \quad \{\ \ map\ sum \cdot concat = concat \cdot map\ (map\ sum)\ \ \}\\ & max \cdot concat \cdot map\ (map\ sum) \cdot map\ inits \cdot tails\\ = & \quad \{\ \ max \cdot concat = max \cdot map\ max\ \ \}\\ & max \cdot map\ max \cdot\\ & \qquad map\ (map\ sum) \cdot map\ inits \cdot tails\\ = & \quad \{\ \ map\ f \cdot map\ g = map\ (f \cdot g)\ \ \}\\ & max \cdot map\ (max \cdot map\ sum \cdot inits) \cdot tails.\end{array}$$

Thus we have

$$mss = max \cdot map\ mps \cdot tails.$$

To compute the best segment-sum, we compute the best prefix-sum for each suffix.

Since *mps* runs in linear time, the definition of *mss* above still runs in $O(n^2)$ time. However, there is a useful "scan lemma" saying that $map\ f \cdot tails$ can be compute efficiently, if f has the form:

```
let rec f = function
  | [] → e
  | x :: xs → g x (f xs)
```

(that is, if f is an instance of a *foldr_right*, an important concept we unfortunately cannot cover yet). The function *mps* fits the pattern if we let $e = 0$ and $g\ x\ y = 0 \uparrow (x + y)$.

Let $scan = map\ f \cdot tails$. To derive the scan lemma we will need a property that

$$hd\ (tails\ xs) = xs,$$

whose proof is easy. We try to construct an inductive definition of *scan*. The base case $scan\ [\,] = [e]$ is easy. For the inductive case:

$$\begin{array}{ll} & scan\ (x :: xs) \\ = & map\ f\ (tails\ (x :: xs)) \\ = & map\ f\ ((x :: xs) :: tails\ xs) \\ = & f\ (x : xs) :: map\ f\ (tails\ xs) \\ = & g\ x\ (f\ xs) :: map\ f\ (tails\ xs) \\ = & \{\ xs = hd\ (tails\ xs), \text{ thus} \\ & \quad f\ xs = hd\ (map\ f\ (tails\ xs))\ \} \\ & \textbf{let}\ ys = map\ f\ (tails\ xs) \\ & \textbf{in}\ g\ x\ (hd\ ys) :: ys \\ = & \textbf{let}\ ys = scan\ xs \\ & \textbf{in}\ g\ x\ (hd\ ys) :: ys. \end{array}$$

Thus we have shown that

$$mss = max \cdot scan,$$

where *scan* is given by

```
let rec scan = function
  | [] → [0]
  | x :: xs → let ys = scan xs in
                0 ↑ (x + hd ys) : ys.
```

You may compare that with the imperative algorithm you may know.

C OCaml Cheatsheet

Adapted from David Matuszek, A Concise Introduction to Objective Caml.
`http://www.csc.villanova.edu/~dmatusze/resources/ocaml/ocaml.html`

Primitive types

There are several primitive types in OCaml; the following table gives the most important ones.

Primitive type	Examples	Notes
`int`	`42, -17, 0x00FF, 0o77, 0b1101`	`-` is used for unary minus; there is no unary `+`. `0x` or `0X` starts a hexadecimal number; `0o` or `0O` starts an octal number; and `0b` or `0B` starts a binary number
`float`	`0.0, -5.3, 1.7e14, 1.7e+14, 1e-10`	Can't start with a decimal point
`bool`	`true, false`	These are the only bool values.
`string`	`"", "One\nTwo"`	`"\n"` is newline, `"\t"` is tab, `"\\" is backslash`
`char`	`'a', '\n'`	Single quotes for chars, double quotes for strings.
`unit`	`()`	This is a value. It is the only value of its type, and is often used when the value isn't important (much like void in C).

There are three families of constructed types in OCaml: lists, tuples, and functions.

Standard bool operators

<table>
<tr><th>Function</th><th>Examples</th><th>Notes</th></tr>
<tr><td><code>not : bool -> bool</code></td><td>n̂ot true, not (i = j)</td><td>(Prefix) Unary negation.</td></tr>
<tr><td><code>&& : bool * bool -> bool</code></td><td><code>(i = j) && (j = k)</code></td><td>(Infix, left associative) Conjunction, with short-circuit evaluation.</td></tr>
<tr><td><code>|| : bool * bool -> bool</code></td><td><code>(i = j) || (j = k)</code></td><td>(Infix, left associative) Disjunction, with short-circuit evaluation.</td></tr>
</table>

Standard arithmetic operators on integers

Function	Examples	Notes
`- : int -> int`	`-5, -limit`	(Prefix) Unary negation.
`* : int * int -> int`	`2 * limit`	(Infix, left associative) Multiplication; operands and result are all ints.
`/ : int * int -> int`	`7 / 3, score / average`	(Infix, left associative) Division; truncates fractional part.
`mod : int * int -> int`	`limit mod 2`	(Infix, left associative) Modulus; result has sign of first operand.
`+ : int * int -> int`	`2 + 2, limit + 1`	(Infix, left associative) Addition.
`- : int * int -> int`	`2 - 2, limit - 1`	(Infix, left associative) Subtraction.
`abs : int -> int`	`abs (-5)`	(Prefix) Absolute value.

Standard arithmetic operators on real numbers

Function	Examples	Notes
`-. : float -> float`	`-1e10, -average`	(Prefix) Unary negation.
`*. : float *. float -> float`	`3.1416 *. r *. r`	(Infix, left associative) Multiplication; operands and result are all real numbers.
`/. : float * float -> float`	`7.0 /. 3.5`	(Infix, left associative) Division of real numbers.
`+. : float * float -> float`	`score +. 1.0`	(Infix, left associative) Addition of real numbers.
`-. : float * float -> float`	`score -. 1.0`	(Infix, left associative) Subtraction of real numbers.
`** : float *. float -> float`	`15.5 ** 2.0`	(Infix, right associative) Exponentiation.
`sqrt : float -> float`	`sqrt 8.0`	(Prefix) Square root.
`ceil : float -> float`	`ceil 9.5`	Round up to nearest integer (but result is still a real number).
`floor : float -> float`	`floor 9.5`	Round down to nearest integer (but result is still a real number).

Besides, we have the usual transcendental functions: `exp, log, log10, cos, sin, tan, acos,` of type `float -> float`.

Coercions

Function	Notes
`float : int -> float`	Convert integer to real.
`truncate : float -> int`	Fractional part is discarded.
`int_of_char : char -> int`	ASCII value of character.
`char_of_int : int -> char`	Character corresponding to ASCII value; argument must be in range 0..255.
`int_of_string : string -> int`	Convert string to integer.
`string_of_int : int -> string`	Convert integer to string.
`float_of_string : string -> float`	Convert string to float.
`string_of_float : float -> string`	Convert float to string.
`bool_of_string : string -> bool`	Convert string to bool.
`string_of_bool : bool -> string`	Convert bool to string.

Comparisons

Function	Notes
`< : 'a * 'a -> bool`	Less than. a' can be int, float, char, or string.
`<= : 'a * 'a -> bool`	Less than or equal to. a' can be int, float, char, or string.
`= : 'a * 'a -> bool`	Equals. a' can be int, char, or string, but not float.
`<> : 'a * 'a -> bool`	Not equal. a' can be int, char, or string, but not float.
`>= : 'a * 'a -> bool`	Greater than or equal to. a' can be int, float, char, or string.
`> : 'a * 'a -> bool`	Greater than. a' can be int, float, char, or string.
`== : 'a -> 'a -> bool`	Physical equality; meaning is somewhat implementation-dependent.
`!= : 'a -> 'a -> bool`	Physical inequality; meaning is somewhat implementation-dependent.
`max : 'a -> 'a -> 'a`	Returns the larger of the two arguments.
`min : 'a -> 'a -> 'a`	Returns the smaller of the two arguments.