GADTs in OCaml

Functional Conf, Bangalore, India

Anmol Sahoo, IIT Madras

About Me

- Project associate at IIT Madras
- Working on the OCaml compiler
- Guided by [KC Sivaramakrishanan (kcsrk.info)](kcsrk.info) (All slides thanks to him as well!)

Multicore OCaml at IIT Madras

- Working on upstreaming the multicore compiler for OCaml
- Lots of work on the compiler as well as application libraries
- Good mix of theoretical and practical work

We are hiring - Hiring Page (http://kcsrk.info/ocaml/multicore/job/2019/09/16/1115-multicore-job/)

Agenda

- Introduction to OCaml syntax - let bindings, functions and pattern matching
- Algebraic Datatypes in OCaml
- *Generalized* Algebraic Datatypes (GADTs)

But first, Why GADTs?

- Type safety - Not **just** the typed interpreter, practical examples
  - Safely typed GraphQL in OCaml (https://andreas.github.io/2017/11/29/type-safe-graphql-with-ocaml-part-1/)
- Performance
  - Why GADTs matter for performance (https://blog.janestreet.com/why-gadts-matter-for-performance/)
- Generic programming
  - Generic Programming in OCaml (https://arxiv.org/pdf/1812.11665.pdf)

Syntax Primer

Values in OCaml

In [ ]:

```
42
```

In [ ]:

```
"Hello"
```

In [ ]:

```
3.1415
```

- Observe that the values have
  - static semantics: types `int` , `string` , `float` .
  - dynamic semantics: the value itself.

Type Inference and annotation

- OCaml compiler **infers** types
  - Compilation fails with type error if it can't
  - Hard part of language design: guaranteeing compiler can infer types when program is correctly written
- You can manually annotate types anywhere – Replace `e` with `(e : t)`
  - Useful for resolving type errors

In [ ]:

```
(42.4 : float)
```

More values

OCaml also support other values. See [manual (https://caml.inria.fr/pub/docs/manual-ocaml/values.html)](https://caml.inria.fr/pub/docs/manual-ocaml/values.html).

In [ ]:

```
()
```

In [ ]:

```
(1,"hello", true, 3.4)
```

In [ ]:

```
[1;2;3]
```

In [ ]:

```
[|1;2;3|]
```

Static vs Dynamic distinction

Static typing helps catch lots errors at compile time.

Which of these is static error?

In [ ]:

```
23 = 45.0
```

In [ ]:

```
23 = 45
```

If expression

```
if e1 then e2 else e3
```

- **Static Semantics:** If `e1` has type `bool`, and `e2` has type `t2` and `e3` has type `t2` then `if e1 then e2 else e3` has type `t2`.
- **Dynamic Semantics:** If `e1` evaluates to true, then evaluate `e2`, else evaluate `e3`

In [ ]:

```
if 32 = 31 then "Hello" else "World"
```

In [ ]:

```
if true then 13 else 13.4
```

Let expression

```
let x = e1 in e2
```

- `x` is an identifier
- `e1` is the binding expression
- `e2` is the body expression
- `let x = e1 in e2` is itself an expression

In [ ]:

```
let x = 5 in x + 5
```

In [ ]:

```
let x = 5 in
let y = 10 in
x + y
```

In [ ]:

```
let x = 5 in
let x = 10 in
x
```

Scopes & shadowing

```
let x = 5 in
let x = 10 in
x
```

is parsed as

```
let x = 5 in
(let x = 10 in
 x)
```

- Importantly, `x` is not mutated; there are two `x` s in different **scopes**.
- Inner definitions **shadow** the outer definitions.

In [ ]:

```
let x = 5 in
let y =
  let x = 10 in
  x
in
x+y
```

Functions

In [ ]:

```
fun x -> x + 1
```

The function type `int -> int` says that it takes one argument of type `int` and returns a value of type `int` .

Function scope

The function body can refer to any variables in scope.

In [ ]:

```
let foo =
  let y = 10 in
  let x = 5 in
  fun z -> x + y + z
```

Functions are values

Can use them *anywhere* we can use values:

- Functions can **take** functions as arguments
- Functions can **return** functions as arguments

As you will see, this is an incredibly powerful language feature.

Function application

The syntax is

```
e0 e1 ... en
```

- No parentheses necessary

Function Application Evaluation

```
e0 e1 ... en
```

- Evaluate `e0 ... en` to values `v0 ... vn`
- Type checking will ensure that `v0` is a function `fun x1 ... xn -> e`
- Substitute `vi` for `xi` in `e` yielding new expression `e'`
- Evaluate `e'` to a value `v`, which is result

Function Application

In [ ]:

```
(fun x -> x + 1) 1
```

In [ ]:

```
(fun x y z -> x + y + z) 1 2 3
```

The above function is syntactic sugar for

In [ ]:

```
(fun x -> fun y -> fun z -> x + y + z) 1 2 3
```

Multi-argument functions do not exist!

Function definition

We can name functions using `let`.

```
let succ = fun x -> x + 1
```

which is semantically equivalent to

```
let succ x = x + 1
```

You'll see the latter form more often.

Function definition

In [ ]:

```
let succ x = x + 1
```

In [ ]:

```
succ 10
```

Function definition

In [ ]:

```
let add x y = x + y
```

In [ ]:

```
let add = fun x -> fun y -> x + y
```

In [ ]:

```
add 5 10
```

Partial Application

```
(fun x y z -> x + y + z) 1
```

returns a function

```
(fun y z -> 1 + y + z)
```

In [ ]:

```
let foo = (fun x y z -> x + y + z) 1
```

In [ ]:

```
foo 2 3
```

Partial Application

A more useful partial application example is defining `succc` and `pred` functions from `add` .

In [ ]:

```
let succ = add 1
let pred = add (-1)
```

In [ ]:

```
succ 10
```

In [ ]:

```
pred 10
```

Recursive Functions

Recursive functions can call themselves. The syntax for recursive function definition is:

```
let rec foo x = ...
```

Notice the `rec` key word.

Recursive Functions

In [ ]:

```
let rec sum_of_first_n n =
  if n <= 0 then 0
  else n + sum_of_first_n (n-1)
```

In [ ]:

```
sum_of_first_n 5
```

Mutually recursive functions

In [ ]:

```
let rec even n =
  if n = 0 then true
  else odd (n-1)

and odd n =
  if n = 0 then false
  else even (n-1)
```

In [ ]:

```
odd 44
```

Data Types

Type aliases

OCaml support the definition of aliases for existing types. For example,

In [ ]:

```
type int_float_pair = int * float
```

In [ ]:

```
let x = (10, 3.14)
```

In [ ]:

```
let y : int_float_pair = x
```

Records

- Records in OCaml represent a collection of named elements.
- A simple example is a point record containing x, y and z fields:

In [ ]:

```
type point = {
  x : int;
  y : int;
  z : int;
}
```

Records: Creation and access

We can create instances of our point type using `{ ... }`, and access the elements of a point using the `.` operator:

In [ ]:

```
let origin = { y = 0; x = 0;z = 0 }

let get_y (r : point) = r.y
```

Product Types

- Records and tuples are known as **product types**.
  - Each value of a product type includes all of the types that constitute the product.

  ```
  type person_r = {name: string; age: int; height: float}
  type person_t = string * int * float
  ```

- Records are indexed by *names* whereas *tuples* are indexed by positions (1st, 2nd, etc.).

Sum Types a.k.a Variants

The type definition syntax is:

```
type t =
| C1 of t1
| C2 of t2
| C3 of t2
| ...
```

- C1, C2, C2 are known as constructors
- t1, t2 and t3 are optional data carried by constructor
- Also known as **Algebraic Data Types**

In [ ]:

```
type color =
  | Red
  | Green
  | Blue
```

In [ ]:

```
let v = (Green , Red)
```

In [ ]:

```
type point = {x : int; y : int}

type shape =
  | Circle of point * float (* center, radius *)
  | Rect of point * point   (* lower-left, upper-right *)
  | ColorPoint of point * color
```

In [ ]:

```
Circle ({x=4;y=3}, 2.5)
```

In [ ]:

```
Rect ({x=3;y=4}, {x=7;y=9})
```

Recursive variant types

Let's define an integer list

In [ ]:

```
type intlist =
  | INil
  | ICons of int * intlist
```

In [ ]:

```
ICons (1, ICons (2, ICons (3, INil)))
```

- `Nil` and `Cons` originate from Lisp.

String List

```
type stringlist =
  | SNil
  | Scons of string * stringlist
```

- Now what about `pointlist`, `shapelist`, etc?

Parameterized Variants

In [ ]:

```
type 'a lst =
    Nil
  | Cons of 'a * 'a lst
```

In [ ]:

```
Cons (1, Cons (2, Nil))
```

In [ ]:

```
Cons ("Hello", Cons("World", Nil))
```

Type Variable

- **Variable**: name standing for an unknown value
- **Type Variable**: name standing for an unknown type

- Java example is `List<T>`

- OCaml syntax for type variable is a single quote followed by an identifier
  - `'foo`, `'key`, `'value`
- Most often just `'a`, `'b`.
  - Pronounced "alpha", "beta" or "quote a", "quote b".

Polymorphism

- The type `'a lst` that we had defined earlier is a **polymorphic data type**.
- poly = many, morph = change.
- write functionality that works for many data types.
- Related to Java Generics and C++ template instantiation.
- In `'a lst`, `lst` is known as a **type constructor**.
  - constructs types such as `int lst`, `string lst`, `shape lst`, etc.

OCaml built-in lists are just variants

OCaml effectively codes up lists as variants:

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

- `[]` and `::` are constuctors.
- Just a bit of syntactic magic to use `[]` and `::` as constructors rather than alphanumeric identifiers.

In [ ]:

```
[]
```

In [ ]:

```
1::2::[]
```

Pattern Matching

Pattern Matching

- Pattern matching is data deconstruction
  - Match on the *shape* of data
  - Extract part(s) of data

Syntax

```
match e with
| p1 -> e1
| p2 -> e2
...
| pn -> en
```

- p1 ... pn are patterns.

Pattern Matching on Lists

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

- For lists, the patterns allowed follow from the constructors
  - The pattern `[]` matches the value `[]`.
  - The patterh `h::t`
    - matches `2::[]`, binding `h` to `2` and `t` to `[]`.
    - matches `2::3::[]`, binding `h` to `2` and `t` to `3::[]`.
  - The pattern `_` is a **wildcard pattern** and matches anything.

In [ ]:

```
let list_status l =
  match l with
  | [] -> print_endline "The list is empty"
  | h::t -> Printf.printf "The list is non-empty. Head = %d\n%!" h
```

In [ ]:

```
list_status []
```

In [ ]:

```
list_status [1;2;3]
```

In [ ]:

```
list_status (2::[3;4])
```

Advantages of pattern matching

1. You cannot forget to match a case (Exhaustivity warning)

In [ ]:

```
let list_status l =
  match l with
  | [] -> print_endline "The list is empty"
  | h1::h2::t -> Printf.printf "The list is non-empty. 2nd element = %d\n%!" h2
```

Advantages of pattern matching

1. You cannot forget to match a case (Exhaustivity warning)
2. You cannot duplicate a case (Unused case warning)

In [ ]:

```
let list_status l =
  match l with
  | [] -> print_endline "The list is empty"
  | h::t -> Printf.printf "The list is non-empty. Head = %d\n%!" h
  | h1::h2::t -> Printf.printf "The list is non-empty. 2nd element = %d\n%!" h2
```

Length of list (tail recursive)

In [ ]:

```
let rec length' l acc =
  match l with
  | [] -> acc
  | h::t -> length' t (1+acc)

let length l = length' l 0
```

In [ ]:

```
length [1;2;3;4]
```

Match ordering

The patterns are matched in the order that they are written down.

In [ ]:

```
let is_empty l =
  match l with
  | _ -> false
  | [] -> true
```

nth

Implement indexing into the list

In [ ]:

```
let rec nth l n =
  match (l, n) with
  | (hd::_, 0) -> Some hd
  | (hd::tl, n) -> nth tl (n-1)
  | _ -> None
```

In [ ]:

```
nth [1;2;3] 4
```

Nested Matching

In [ ]:

```
type color = Red | Green | Blue

type point = {x : int; y : int}

type shape =
  | Circle of point * float (* center, radius *)
  | Rect of point * point   (* lower-left, upper-right *)
  | ColorPoint of point * color
```

Nested Matching

Is the first shape in a list of shapes a red point?

In [ ]:

```
let is_hd_red_circle l =
  match l with
  | ColorPoint(_,Red)::_ -> true
  | _ -> false
```

Nested Matching

Print the coordinates if the point is green.

In [ ]:

```
let rec print_green_point l =
  match l with
  | [] -> ()
  | ColorPoint({x;y}, Green)::tl ->
      Printf.printf "x = %d y = %d\n%!" x y;
      print_green_point tl
  | _::tl -> print_green_point tl
```

In [ ]:

```
print_green_point [Rect ({x=1;y=1},{x=2;y=2});
                   ColorPoint ({x=0;y=0}, Green);
                   Circle ({x=1;y=3}, 5.4);
                   ColorPoint ({x=4;y=6}, Green)]
```

Generalized Algebraic Data Types

Simple language

Consider this simple language of integers and booleans

In [ ]:

```
type value =
  | Int of int
  | Bool of bool

type expr =
  | Val of value
  | Plus of expr * expr
  | Mult of expr * expr
  | Ite of expr * expr * expr
```

Evaluator for the simple language

We can write a simple evaluator for this language

In [ ]:

```
let rec eval : expr -> value =
  fun e -> match e with
  | Val (Int i) -> Int i
  | Val (Bool i) -> Bool i
  | Plus (e1, e2) ->
    let Int i1, Int i2 = eval e1, eval e2 in
    Int (i1 + i2)
  | Mult (e1, e2) ->
    let Int i1, Int i2 = eval e1, eval e2 in
    Int (i1 * i2)
  | Ite (p,e1,e2) ->
    let Bool b = eval p in
    if b then eval e1 else eval e2
```

Evaluator for the simple language

- The compiler warns that programs such as `true + 10` is not handled.
  - Our evaluator gets **stuck** when it encouters such an expression.

In [ ]:

```
eval (Plus (Val (Bool true), Val (Int 10)))
```

- We need **Types**
  - Well-typed programs do not get stuck!

Phantom types

- We can add types to our values using a technique called **phantom types**

In [ ]:

```
type 'a value =
  | Int of int
  | Bool of bool
```

- Observe that `'a` only appears on the LHS.
  - This `'a` is called a phantom type variable.
- What is this useful for?

Typed expression language

We can add types to our expression language now using phantom type

In [ ]:

```
type 'a expr =
  | Val of 'a value
  | Plus of int expr * int expr
  | Mult of int expr * int expr
  | Ite of bool expr * 'a expr * 'a expr
```

Typed expression language

Assign concerte type to the phantom type variable `'a`.

In [ ]:

```
(* Quiz: What types are inferred without type annotations? *)
let mk_int i : int expr = Val (Int i)
let mk_bool b : bool expr = Val (Bool b)
let plus e1 e2 : int expr = Plus (e1, e2)
let mult e1 e2 : int expr = Mult (e1, e2)
```

Benefit of phantom types

In [ ]:

```
let i = Val (Int 0);;
let i' = mk_int 0;;

let b = Val (Bool true);;
let b' = mk_bool true;;

let p = Plus (i,i);;
let p' = plus i i;;
```

Benefit of phantom types

We no longer allow ill-typed expression if we use the helper functions.

In [ ]:

```
plus (mk_bool true) (mk_int 10)
```

Typed evaluator

We can write an evaluator for this language now.

Let's use the same evaluator as the earlier one.

In [ ]:

```
let rec eval : 'a expr -> 'a value =
  fun e -> match e with
  | Val (Int i) -> Int i
  | Val (Bool i) -> Bool i
  | Plus (e1, e2) ->
    let Int i1, Int i2 = eval e1, eval e2 in
    Int (i1 + i2)
  | Mult (e1, e2) ->
    let Int i1, Int i2 = eval e1, eval e2 in
    Int (i1 * i2)
  | Ite (p,e1,e2) ->
    let Bool b = eval p in
    if b then eval e1 else eval e2
```

Typed evaluator

- We see a $\color{red}{\text{type error}}$.
- OCaml by default expects the function expression at the recursive call position to have the same type as the outer function.
- This need not be the case if the recursive function call is at different types.
  - `eval (p : int expr)` and `eval (p : bool expr)`.

Polymorphic recursion.

- In order to allow this, OCaml supports polymorphic recursion (aka Milner-Mycroft typeability)
  - Robin Milner co-invented type infererence + polymorphism that we use in OCaml.

Fixing the interpreter with polymorphic recursion

`type a` is known as **locally abstract type**.

In [ ]:

```
let rec eval : type a. a expr -> a value =
  fun e -> match e with
  | Val (Int i) -> Int i
  | Val (Bool i) -> Bool i
  | Plus (e1, e2) ->
    let Int i1, Int i2 = eval e1, eval e2 in
    Int (i1 + i2)
  | Mult (e1, e2) ->
    let Int i1, Int i2 = eval e1, eval e2 in
    Int (i1 * i2)
  | Ite (p,e1,e2) ->
    let Bool b = eval p in
    if b then eval e1 else eval e2
```

Errors gone, but warning remains

- Compiler still warns us that there are unhandled cases in pattern matches
- But haven't we added types to the expression language?
- Observe that `mk_int i = Val (Int i)` is just convention.
  - You can still write ill-typed expression by directly using the constructors.

Errors gone, but warning remains

In [ ]:

```
eval (Plus (Val (Bool true), Val (Int 10)))
```

- Here, `Bool true` is inferred to have the type `int value` .
  - Need a way to inform the compiler that `Bool true` has type `bool value` .

Generalized Algebraic Data Types

GADTs allow us to **refine** the return type of the data constructor.

In [ ]:

```
type 'a value =
  | Int : int -> int value
  | Bool : bool -> bool value

type 'a expr =
  | Val : 'a value -> 'a expr
  | Plus : int expr * int expr -> int expr
  | Mult : int expr * int expr -> int expr
  | Ite : bool expr * 'a expr * 'a expr -> 'a expr
```

Evaluator remains the same

Observe that the warnings are also gone!

In [ ]:

```
let rec eval : type a. a expr -> a value =
  fun e -> match e with
  | Val (Int i) -> Int i
  | Val (Bool i) -> Bool i
  | Plus (e1, e2) ->
    let Int i1, Int i2 = eval e1, eval e2 in
    Int (i1 + i2)
  | Mult (e1, e2) ->
    let Int i1, Int i2 = eval e1, eval e2 in
    Int (i1 * i2)
  | Ite (p,e1,e2) ->
    let Bool b = eval p in
    if b then eval e1 else eval e2
```

Absurd expressions are ill-typed

In [ ]:

```
eval (Plus (Val (Bool true), Val (Int 10)))
```

Absurd types

GADTs don't prevent you from instantiating **absurd** types. Consider

```
type 'a value =
  | Int : int -> int value
  | Bool : bool -> bool value
```

In [ ]:

```
type t = string value
```

- There is no term with type `string value`
- We will ignore such types.

GADTs are very powerful!

- Allows **refining return types** and introduce **existential types** (to be discussed).
- Some uses
  - Typed domain specific languages
    - The example that we just saw...
  - (Lightweight) dependently typed programming
    - Enforcing **shape properties** of data structures
  - Generic programming
    - Implementing functions like `map` and `fold` operate on the shape of the data **once and for all**!

GADT examples

- Units of measure
- Abstract (existential) types - encoding first-class modules
- Generic programming - encoding tuples
- Shape properties - length-indexed lists

Units of measure

- In 1999, $125 million [mars climate orbiter (https://en.wikipedia.org/wiki/Mars_Climate_Orbiter)](https://en.wikipedia.org/wiki/Mars_Climate_Orbiter) was lost due to units of measurement error
  - Lockheed Martin used Imperial and NASA used Metric
  - Use GADTs to avoid such errors, but still host both units of measure in the same program

Units of measure

In [ ]:

```
type kelvin
type celcius
type farenheit

type _ temp =
  | Kelvin : float -> kelvin temp
  | Celcius : float -> celcius temp
  | Farenheit : float -> farenheit temp
```

Units of measure

In [ ]:

```
let add_temp : type a. a temp -> a temp -> a temp =
  fun a b -> match a,b with
  | Kelvin a, Kelvin b -> Kelvin (a+.b)
  | Celcius a, Celcius b -> Celcius (a+.b)
  | Farenheit a, Farenheit b -> Farenheit (a+.b)
```

In [ ]:

```
add_temp (Kelvin 20.23) (Kelvin 30.5)
```

In [ ]:

```
add_temp (Kelvin 20.23) (Celcius 12.3)
```

Abstract types

- GADTs also introduce abstract types (aka **existential type**).

In [ ]:

```
type t = Pack : 'a -> t
```

- Observe that the `'a` does not appear on the RHS.
  - `'a` is the **existential type**.
  - Given a value `Pack x` of type `t`, we know nothing about the type of `x` except that such a type exists.
- Compare with `Some x` which has type `'a t`, where `x` is of type `'a`.

Abstract List

With GADTs you can create list that contains values of different types.

In [ ]:

```
[Pack 10; Pack "Hello"; Pack true]
```

- This particular list isn't useful
  - Given `Pack v`, we only know that `v` has some type `'a`.
  - We do not have any useful operations on values of type `'a`; it is too polymorphic.

Existential list : showable

Here is a more useful heterogeneous list: List of printable values.

In [ ]:

```
type showable = Showable : 'a * ('a -> string) -> showable
```

In [ ]:

```
let l = [Showable (10, string_of_int); Showable ("Hello", fun x -> x);
         Showable (3.14, string_of_float)]
```

In [ ]:

```
List.map (fun (Showable (v,show)) -> show v) l
```

Encoding Tuples

We can encode OCaml-like tuples using GADTs.

In [ ]:

```
type u = | (* uninhabited type *)

type _ hlist =
  | Nil : u hlist
  | Cons : 'a * 'b hlist -> ('a * 'b) hlist
```

In [ ]:

```
let l = Cons (10, Cons (false, Cons (10.4, Nil)))
```

Encoding Pairs : Accessor Functions

In [ ]:

```
let fst : ('a * _) hlist -> 'a = fun (Cons (x,_)) -> x
let snd : (_ * ('a * _)) hlist -> 'a = fun (Cons (_,Cons(x,_))) -> x
let trd : (_ * (_ * ('a * _))) hlist -> 'a = fun (Cons(_,Cons (_,Cons(x,_)))) -> x
```

Encoding Pairs : Accessor Functions

In [ ]:

```
trd (Cons (10, Cons (true, Cons(10.5, Nil))))
```

In [ ]:

```
trd (Cons (true, Cons(10.5, Nil)))
```

Length-indexed lists

Some of the list function in the OCaml list library as quite unsatisfying.

In [ ]:

```
List.hd []
```

In [ ]:

```
List.tl []
```

Morever, these errors caught at runtime

In [ ]:

```
let get_head x = List.hd x
```

In [ ]:

```
get_head []
```

Length indexed lists

- Let's implement our own list type which will statically catch these errors.
- The idea is to encode the **length** of the list in the **type** of the list.
    - Use our encoding of church numerals from lambda calculus.

Church numerals in OCaml types

In [ ]:

```
type z = Z
type 'n s = S : 'n -> 'n s
```

In [ ]:

```
S (S Z)
```

Length indexed list

In [ ]:

```
type (_,_) list =
  | Nil  : ('a, z ) list
  | Cons : 'a * ('a,'n) list -> ('a, 'n s) list
```

In [ ]:

```
Nil;;
Cons(0,Nil);;
Cons(0,Cons(1,Nil));;
```

Safe hd and tl

Define the function hd and tl such that they can only be applied to non-empty lists.

In [ ]:

```
let hd (l : ('a,'n s) list) : 'a =
  let Cons (v,_) = l in
  v
```

In [ ]:

```
hd (Cons (1, Nil))
```

In [ ]:

```
hd Nil
```

Safe hd and tl

Define the function hd and tl such that they can only be applied to non-empty lists.

In [ ]:

```
let hd (l : ('a,'n s) list) : 'a =
  let Cons (x,_) = l in
  x
```

- Observe that OCaml does not complain about Nil case not handled.
    - Does not apply since l is non-empty!
    - GADTs allow the compiler to refute cases statically
        - Generate more efficient code!

Safe hd and tl

In [ ]:

```
let tl (l : ('a,'n s) list) : ('a, 'n) list =
  let Cons (_,xs) = l in
  xs
```

In [ ]:

```
tl (Cons (0, Cons(1,Nil)));;
tl (Cons (0, Nil));;
```

List map

map is length preserving

In [ ]:

```
let rec map : type n. ('a -> 'b) -> ('a, n) list -> ('b, n) list =
  fun f l ->
    match l with
    | Nil -> Nil
    | Cons (x,xs) -> Cons(f x, map f xs)
```

Non length-preserving map rejected

In [ ]:

```
let rec map' : type n. 'p -> ('p -> 'q) -> ('p, n) list -> ('q, n) list =
  fun a f l ->
     match l with
          | Nil -> Cons (f a, Nil)
          | Cons (x,xs) -> Cons(f x, map' a f xs)
```

Trees

Here is an unconstrained tree data type:

In [ ]:

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

In [ ]:

```
Tree (Empty, 1, Tree (Empty, 2, Tree (Empty, 3, Empty)));; (* Right skewed *)
Tree (Tree (Tree (Empty, 3, Empty), 2, Empty), 1, Empty);; (* Left skewed *)
Tree (Tree (Tree (Empty, 3, Empty), 2, Tree (Empty, 3, Empty)),
      1,
      Tree (Tree (Empty, 3, Empty), 2, Tree (Empty, 3, Empty))) (* Perfectly balanced tree *)
```

Tree operations

In [ ]:

```
let rec depth t = match t with
  | Empty -> 0
  | Tree (l,_,r) -> 1 + max (depth l) (depth r)
```

In [ ]:

```
let top t = match t with
  | Empty -> None
  | Tree (_,v,_) -> Some v
```

`swivel` is mirror image of the tree

In [ ]:

```
let rec swivel t = match t with
  | Empty -> Empty
  | Tree (l,v,r) -> Tree (swivel r, v, swivel l)
```

Perfectly balanced tree using GADTs

In [ ]:

```
type ('a,_) gtree =
  | EmptyG : ('a,z) gtree
  | TreeG  : ('a,'n) gtree * 'a * ('a,'n) gtree -> ('a,'n s) gtree
```

In [ ]:

```
TreeG (TreeG (TreeG (EmptyG, 3, EmptyG), 2, TreeG (EmptyG, 3, EmptyG)),
       1,
       TreeG (TreeG (EmptyG, 3, EmptyG), 2, TreeG (EmptyG, 3, EmptyG)))
```

Operations on `gtree`

In [ ]:

```
let rec depthG : type n. ('a,n) gtree -> int =
  fun t -> match t with
  | EmptyG -> 0
  | TreeG (l,_,_) -> 1 + depthG l
```

In [ ]:

```
let topG : ('a, 'n s) gtree -> 'a =
  fun t -> let TreeG(_,v,_) = t in v
```

In [ ]:

```
let rec swivelG : type n.('a,n) gtree -> ('a,n) gtree =
fun t -> match t with
    EmptyG -> EmptyG
  | TreeG (l,v,r) -> TreeG (swivelG r, v, swivelG l)
```

Zipping perfect trees

In [ ]:

```
let rec zipTree :
  type n.('a,n) gtree -> ('b,n) gtree -> ('a * 'b,n) gtree =
  fun x y -> match x, y with
      EmptyG, EmptyG -> EmptyG
    | TreeG (l,v,r), TreeG (m,w,s) ->
      TreeG (zipTree l m, (v,w), zipTree r s)
```

Thank you.

Questions?