Generalized Algebraic Data Types

CS3100 Fall 2019

Simple language

Consider this simple language of integers and booleans

In [1]:

```
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
```

Out[1]:

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

Out[1]:

```
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 [2]:

```
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
```

```
File "[2]", line 6, characters 4-62:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
((Int _, Bool _)|(Bool _, _))
File "[2]", line 9, characters 4-62:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
((Int _, Bool _)|(Bool _, _))
File "[2]", line 12, characters 4-61:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Int _
```

Out[2]:

```
val eval : expr -> value = <fun>
```

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 [3]:

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

```
Exception: Match_failure ("[2]", 6, 4).
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

- 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 [4]:

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

Out[4]:

```
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 [5]:

```
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
```

Out[5]:

```
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 [6]:

```
(* 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)
```

Out[6]:

```
val mk_int : int -> int expr = <fun>
```

Out[6]:

```
val mk_bool : bool -> bool expr = <fun>
```

Out[6]:

```
val plus : int expr -> int expr -> int expr = <fun>
```

Out[6]:

```
val mult : int expr -> int expr -> int expr = <fun>
```

Benefit of phantom types

In [7]:

```
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;;
```

Out[7]:

```
val i : 'a expr = Val (Int 0)
```

Out[7]:

```
val i' : int expr = Val (Int 0)
```

Out[7]:

```
val b : 'a expr = Val (Bool true)
```

Out[7]:

```
val b' : bool expr = Val (Bool true)
```

Out[7]:

```
val p : 'a expr = Plus (Val (Int 0), Val (Int 0))
```

Out[7]:

```
val p' : int expr = Plus (Val (Int 0), Val (Int 0))
```

Benefit of phantom types

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

In [8]:

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

```
File "[8]", line 1, characters 5-19:
Error: This expression has type bool expr
       but an expression was expected of type int expr
       Type bool is not compatible with type int
   1: 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 [9]:

```
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
```

```
File "[9]", line 6, characters 4-62:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
((Int _, Bool _)|(Bool _, _))
File "[9]", line 9, characters 4-62:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
((Int _, Bool _)|(Bool _, _))

File "[9]", line 12, characters 22-23:
Error: This expression has type bool expr
       but an expression was expected of type int expr
       Type bool is not compatible with type int
  11:     | Ite (p,e1,e2) ->
  12:       let Bool b = eval p in
  13:       if b then eval e1 else eval e2
```

Typed evaluator

- We see a 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.
  - Alan Mycroft was my mentor at Cambridge :-)

Fixing the interpreter with polymorphic recursion

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

In [10]:

```
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
```

```
File "[10]", line 6, characters 4-62:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
((Int _, Bool _)|(Bool _, _))
File "[10]", line 9, characters 4-62:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
((Int _, Bool _)|(Bool _, _))
File "[10]", line 12, characters 4-61:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Int _
```

Out[10]:

```
val eval : 'a expr -> 'a value = <fun>
```

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 [11]:

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

```
Exception: Match_failure ("[10]", 6, 4).
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

- 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 [12]:

```
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
```

Out[12]:

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

Out[12]:

```
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 [13]:

```
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
```

Out[13]:

```
val eval : 'a expr -> 'a value = <fun>
```

Absurd expressions are ill-typed

In [14]:

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

```
File "[14]", line 1, characters 18-29:
Error: This expression has type bool value
       but an expression was expected of type int value
       Type bool is not compatible with type int
   1: 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 [15]:

```
type t = string value
```

Out[15]:

```
type t = string value
```

- There is no term with type `string value`
  - Recall from simply typed lambda calculus that such types are known as **uninhabited types**.
- 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 [16]:

```
type kelvin
type celcius
type farenheit

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

Out[16]:

```
type kelvin
```

Out[16]:

```
type celcius
```

Out[16]:

```
type farenheit
```

Out[16]:

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

Units of measure

In [17]:

```
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)
```

Out[17]:

```
val add_temp : 'a temp -> 'a temp -> 'a temp = <fun>
```

In [18]:

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

Out[18]:

```
- : kelvin temp = Kelvin 50.730000000000004
```

In [19]:

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

```
File "[19]", line 1, characters 24-38:
Error: This expression has type celcius temp
       but an expression was expected of type kelvin temp
       Type celcius is not compatible with type kelvin
   1: add_temp (Kelvin 20.23) (Celcius 12.3)
```

Abstract types

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

In [20]:

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

Out[20]:

```
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 [21]:

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

Out[21]:

```
- : t list = [Pack <poly>; Pack <poly>; Pack <poly>]
```

- 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 [22]:

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

Out[22]:

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

In [23]:

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

Out[23]:

```
val l : showable list =
  [Showable (<poly>, <fun>); Showable (<poly>, <fun>);
   Showable (<poly>, <fun>)]
```

In [24]:

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

Out[24]:

```
- : string list = ["10"; "Hello"; "3.14"]
```

GADTs and Modules

The type `type showable = Showable : 'a * ('a -> string) -> showable` is equivalent to

```
module type Showable : sig
  type t
  val value : t
  val show : t -> string
end
```

GADTs and Modules

And the value `Showable (10, string_of_int)` is equivalent to

```
module IntShowable : Showable = struct
  type t = int
  let value = 10
  let show = string_of_int
end
```

Both GADTs and Modules introduce existentials.

First-class modules

- Unlike modules, the GADTs are values.
- `[Showable (10, string_of_int)]` is a list of showable values.
  - Can't do this with the module language we've studied.
  - we will ignore the fact that OCaml has first-class modules for now.

Encoding Tuples

We can encode OCaml-like tuples using GADTs.

In [25]:

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

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

Out[25]:

```
type u = |
```

Out[25]:

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

In [26]:

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

Out[26]:

```
val l : (int * (bool * (float * u))) hlist =
  Cons (10, Cons (false, Cons (10.4, Nil)))
```

Encoding Pairs : Accessor Functions

In [27]:

```
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,
```

Out[27]:

```
val fst : ('a * 'b) hlist -> 'a = <fun>
```

Out[27]:

```
val snd : ('b * ('a * 'c)) hlist -> 'a = <fun>
```

Out[27]:

```
val trd : ('b * ('c * ('a * 'd))) hlist -> 'a = <fun>
```

Encoding Pairs : Accessor Functions

In [28]:

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

Out[28]:

```
- : float = 10.5
```

In [29]:

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

```
File "[29]", line 1, characters 28-31:
Error: This expression has type u hlist
       but an expression was expected of type ('a * 'b) hli
st
       Type u is not compatible with type 'a * 'b
   1: trd (Cons (true, Cons(10.5, Nil)))
```

Length-indexed lists

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

In [30]:

```
List.hd []
```

```
Exception: Failure "hd".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

In [31]:

```
List.tl []
```

```
Exception: Failure "tl".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

Length indexed lists

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 [32]:

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

Out[32]:

```
type z = Z
```

Out[32]:

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

In [33]:

```
S (S Z)
```

Out[33]:

```
- : z s s = S (S Z)
```

Length indexed list

In [34]:

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

Out[34]:

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

In [35]:

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

Out[35]:

```
- : ('a, z) list = Nil
```

Out[35]:

```
- : (int, z s) list = Cons (0, Nil)
```

Out[35]:

```
- : (int, z s s) list = 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 [36]:

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

Out[36]:

```
val hd : ('a, 'n s) list -> 'a = <fun>
```

In [37]:

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

Out[37]:

```
- : int = 1
```

In [38]:

```
hd Nil
```

```
File "[38]", line 1, characters 3-6:
Error: This expression has type ('a, z) list
       but an expression was expected of type ('a, 'b s) li
st
       Type z is not compatible with type 'b s
   1: hd Nil
```

Safe `hd` and `tl`

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

In [39]:

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

Out[39]:

```
val hd : ('a, 'n s) list -> 'a = <fun>
```

- 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 [40]:

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

Out[40]:

```
val tl : ('a, 'n s) list -> ('a, 'n) list = <fun>
```

In [41]:

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

Out[41]:

```
- : (int, z s) list = Cons (1, Nil)
```

Out[41]:

```
- : (int, z) list = Nil
```

List map

`map` is length preserving

In [42]:

```
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)
```

Out[42]:

```
val map : ('a -> 'b) -> ('a, 'n) list -> ('b, 'n) list = <f
un>
```

List.rev

Tricky to implement tail recurive list like:

```
let rec rev l acc =
  match l with
  | Nil -> acc
  | Cons(x,xs) -> rev xs (Cons (x,acc))
```

why?

The type of this function is

```
('a,'n) list -> ('a,'m) list -> ('a, 'm + 'n) list
```

We don't have type-level arithmetic

List.rev

Here is another attempt:

In [43]:

```
let rec app1 : type n. ('a, n) list -> 'a -> ('a, n s) list =
  fun l v ->
    match l with
    | Nil -> Cons (v, Nil)
    | Cons(x,xs) -> Cons (x, app1 xs v)

let rec rev : type n. ('a, n) list -> ('a, n) list =
  fun l ->
    match l with
    | Nil -> Nil
    | Cons(x,xs) -> app1 (rev xs) x
```

Out[43]:

```
val app1 : ('a, 'n) list -> 'a -> ('a, 'n s) list = <fun>
```

Out[43]:

```
val rev : ('a, 'n) list -> ('a, 'n) list = <fun>
```

In [44]:

```
rev (Cons (0, Cons (1, Cons (2, Nil))))
```

Out[44]:

```
- : (int, z s s s) list = Cons (2, Cons (1, Cons (0, Nil)))
```

Type level addition

- We observed that tail recursive list function requires type level addition on church numerals
- OCaml doesn't support type level functions natively
  - But we can construct **proofs** for type level functions.

Type level addition

In [45]:

```
type (_,_,_) plus =
  | PlusZero : (z,'n,'n) plus
  | PlusShift  : ('a, 'b s, 'c s) plus -> ('a s, 'b, 'c s) plus
```

Out[45]:

```
type (_, _, _) plus =
    PlusZero : (z, 'n, 'n) plus
  | PlusShift : ('a, 'b s, 'c s) plus -> ('a s, 'b, 'c s) p
lus
```

- `('a, 'b, 'c) plus` ≡ `'c = 'a + 'b` where `'a, 'b, 'c` are type level church numerals.
- `PlusZero` and `PlusOne` are theorems on numbers.
  - $0 + n = n$
  - $a + (b + 1) = c + 1 \implies (a + 1) + b = c + 1$

Type level addition

In [46]:

```
type (_,_,_) plus =
  | PlusZero : (z,'n,'n) plus
  | PlusShift  : ('a, 'b s, 'c s) plus -> ('a s, 'b, 'c s) plus
```

Out[46]:

```
type (_, _, _) plus =
    PlusZero : (z, 'n, 'n) plus
  | PlusShift : ('a, 'b s, 'c s) plus -> ('a s, 'b, 'c s) p
lus
```

- From a Curry-Howard perspective, we can view a value of type `(a,b,c) plus` as a proof of the proposition that `a + b = c`
- Looked at this way, `plus` and other GADTs are a convenient way of programming with proofs as first-class objects.

Tail recursive reverse

In [47]:

```
let rec rev : type m n o. (m,n,o) plus -> ('a, m) list -> ('a, n) list -
  fun p l acc ->
    match p, l with
    | PlusZero, Nil -> acc
    | PlusShift p', Cons(x,xs) ->  rev p' xs (Cons(x,acc))
```

Out[47]:

```
val rev :
  ('m, 'n, 'o) plus -> ('a, 'm) list -> ('a, 'n) list ->
('a, 'o) list =
  <fun>
```

Tail recursive reverse

In [48]:

```
let proof = PlusShift (PlusShift (PlusShift (PlusZero : (z,z s s s,z s s
```

Out[48]:

```
val proof : (z s s s, z, z s s s) plus =
  PlusShift (PlusShift (PlusShift PlusZero))
```

In [49]:

```
rev proof (Cons (0, Cons (1, Cons (2, Nil)))) Nil
```

Out[49]:

```
- : (int, z s s s) list = Cons (2, Cons (1, Cons (0, Nil)))
```

- We have to construct proof by hand :-(
  - Other FP languages (Haskell, Agda, Idris, F*, etc) construct automatic proofs.
  - Register for **Proofs and Programs** course next sem to learn more!

Trees

Here is an unconstrained tree data type:

In [50]:

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

Out[50]:

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

In [51]:

```
Tree (Empty, 1, Tree (Empty, 2, Tree (Empty, 3, Empty)));; (* Right skew
Tree (Tree (Tree (Empty, 3, Empty), 2, Empty), 1, Empty);; (* Left skewe
Tree (Tree (Tree (Empty, 3, Empty), 2, Tree (Empty, 3, Empty)),
      1,
      Tree (Tree (Empty, 3, Empty), 2, Tree (Empty, 3, Empty))) (* Perfe
```

Out[51]:

```
- : int tree = Tree (Empty, 1, Tree (Empty, 2, Tree (Empty,
3, Empty)))
```

Out[51]:

```
- : int tree = Tree (Tree (Tree (Empty, 3, Empty), 2, Empt
y), 1, Empty)
```

Out[51]:

```
- : int tree =
Tree (Tree (Tree (Empty, 3, Empty), 2, Tree (Empty, 3, Empt
y)), 1,
 Tree (Tree (Empty, 3, Empty), 2, Tree (Empty, 3, Empty)))
```

Tree operations

In [52]:

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

Out[52]:

```
val depth : 'a tree -> int = <fun>
```

In [53]:

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

Out[53]:

```
val top : 'a tree -> 'a option = <fun>
```

`swivel` is mirror image of the tree

In [54]:

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

Out[54]:

```
val swivel : 'a tree -> 'a tree = <fun>
```

Perfectly balanced tree using GADTs

In [55]:

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

Out[55]:

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

In [56]:

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

Out[56]:

```
- : (int, z s s s) gtree =
TreeG (TreeG (TreeG (EmptyG, 3, EmptyG), 2, TreeG (EmptyG,
3, EmptyG)), 1,
 TreeG (TreeG (EmptyG, 3, EmptyG), 2, TreeG (EmptyG, 3, Emp
tyG)))
```

Operations on `gtree`

In [57]:

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

Out[57]:

```
val depthG : ('a, 'n) gtree -> int = <fun>
```

In [58]:

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

Out[58]:

```
val topG : ('a, 'n s) gtree -> 'a = <fun>
```

In [59]:

```
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)
```

Out[59]:

```
val swivelG : ('a, 'n) gtree -> ('a, 'n) gtree = <fun>
```

Zipping perfect trees

In [60]:

```
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)
```

Out[60]:

```
val zipTree : ('a, 'n) gtree -> ('b, 'n) gtree -> ('a * 'b,
'n) gtree = <fun>
```

Fin.

Extra Materials.

Depth indexed tree (unbalanced)

```
type ('a,_) dtree =
  | EmptyD : ('a,z) dtree
  | TreeD : ('a,'m) dtree * 'a * ('a,'n) dtree
            * ('m,'n,'o) max -> ('a,'o s) dtree
```

- The type `('m,'n,'o) max` $~\equiv~$ `max('m,'n) = 'o` on type level church numerals.
  - `('m,'n,'o) max` is **proof** that `'o` is the max of `'m` and `'n`.
- `TreeD` carries a value of type `('m,'n,'o) max`
  - This is **proof carrying code**
  - Given `t = TreeD (l,v,r,p : ('m,'n,'o) max)`, we can prove that `t` has depth `'o s`.

Equality GADT

For defining `max` start with the **Equality GADT**.

The equality GADT is the type:

In [61]:

```
type (_,_) eql = Refl : ('a,'a) eql
```

Out[61]:

```
type (_, _) eql = Refl : ('a, 'a) eql
```

We can only instantiate `Refl` with types that are known to be equal.

In [62]:

```
type t = int
let _ = (Refl : (t,int) eql)
```

Out[62]:

```
type t = int
```

Out[62]:

```
- : (t, int) eql = Refl
```

Equality GADT

`Refl` cannot be instantiated at types known to be different or not known to be equal.

In [63]:

```
let _ = (Refl : (int,string) eql)
```

```
File "[63]", line 1, characters 9-13:
Error: This expression has type (int, int) eql
       but an expression was expected of type (int, string)
eql
       Type int is not compatible with type string
   1: let _ = (Refl : (int,string) eql)
```

In [64]:

```
module M : sig type t end = struct type t = int end
let _ = (Refl : (M.t,int) eql)
```

Out[64]:

```
module M : sig type t end
```

```
File "[64]", line 2, characters 9-13:
Error: This expression has type (M.t, M.t) eql
       but an expression was expected of type (M.t, int) eq
l
       Type M.t is not compatible with type int
   1: module M : sig type t end = struct type t = int end
   2: let _ = (Refl : (M.t,int) eql)
```

`max` type

The following type definitions contain theorems about the `max` function.

In [65]:

```
type (_,_,_) max =
    MaxEq : ('a,'b) eql -> ('a,'b,'a) max
  | MaxFlip : ('a,'b,'c) max -> ('b,'a,'c) max
  | MaxSuc : ('a,'b,'a) max -> ('a s,'b,'a s) max
```

Out[65]:

```
type (_, _, _) max =
    MaxEq : ('a, 'b) eql -> ('a, 'b, 'a) max
  | MaxFlip : ('a, 'b, 'c) max -> ('b, 'a, 'c) max
  | MaxSuc : ('a, 'b, 'a) max -> ('a s, 'b, 'a s) max
```

`max` type examples

In [66]:

```
let m1 = MaxEq (Refl : (z, z) eql)
```

Out[66]:

```
val m1 : (z, z, z) max = MaxEq Refl
```

In [67]:

```
let m2 = MaxSuc m1
```

Out[67]:

```
val m2 : (z s, z, z s) max = MaxSuc (MaxEq Refl)
```

Given a proof that the max of `z` and `z` is `z`, I can prove that the max of `z s` and `z` is `z s`.

`max` function on church numerals

In [68]:

```
let rec max : type a b c.(a,b,c) max -> a -> b -> c
  = fun mx m n -> match mx,m with
      MaxEq Refl , _     -> m
    | MaxFlip mx', _     -> max mx' n m
    | MaxSuc mx' , S m'  -> S (max mx' m' n)
```

Out[68]:

```
val max : ('a, 'b, 'c) max -> 'a -> 'b -> 'c = <fun>
```

In [69]:

```
max (MaxSuc (MaxEq (Refl : (z,z) eql))) (S Z) Z
```

Out[69]:

```
- : z s = S Z
```

`max` function on church numerals

In [70]:

```
max (MaxSuc (MaxEq (Refl : (z,z) eql))) (S Z) Z
```

Out[70]:

```
- : z s = S Z
```

- The `max` function seems a bit silly given that we need to provide the proof.
  - We learn no new information. Proof already tells us that `z s > z`.

- `max` function gets **term** level evidence from the **type** level proof.

Depth Indexed Tree

In [71]:

```
type ('a,_) dtree =
    EmptyD : ('a,z) dtree
  | TreeD : ('a,'m) dtree * 'a * ('a,'n) dtree * ('m,'n,'o) max
    -> ('a,'o s) dtree
```

Out[71]:

```
type ('a, _) dtree =
    EmptyD : ('a, z) dtree
  | TreeD : ('a, 'm) dtree * 'a * ('a, 'n) dtree *
      ('m, 'n, 'o) max -> ('a, 'o s) dtree
```

Operations on `dtree`

In [72]:

```
let rec depthD : type a n.(a,n) dtree -> n = function
    EmptyD -> Z
  | TreeD (l,_,r,mx) -> S (max mx (depthD l) (depthD r))
```

Out[72]:

```
val depthD : ('a, 'n) dtree -> 'n = <fun>
```

In [73]:

```
let topD : type a n.(a,n s) dtree -> a =
  function TreeD (_,v,_,_) -> v
```

Out[73]:

```
val topD : ('a, 'n s) dtree -> 'a = <fun>
```

In [74]:

```
let rec swivelD :
  type a n.(a,n) dtree -> (a,n) dtree = function
    EmptyD -> EmptyD
  | TreeD (l,v,r,m) ->
    TreeD (swivelD r, v, swivelD l, MaxFlip m)
```

Out[74]:

```
val swivelD : ('a, 'n) dtree -> ('a, 'n) dtree = <fun>
```

Adding lambdas to interpreter

Let's add simply typed lambda calculus to our original int & bool expression evaluator.

In [75]:

```
type _ typ =
  | TInt : int typ
  | TBool : bool typ
  | TLam : 'a typ * 'b typ -> ('a -> 'b) typ

type 'a value =
  | VInt : int -> int value
  | VBool : bool -> bool value
  | VLam : ('a value -> 'b value) -> ('a -> 'b) value

and 'a expr =
  | Var : string * 'a typ ->  'a expr
  | App : ('a -> 'b) expr * 'a expr -> 'b expr
  | Lam : string * 'a typ * 'b expr -> ('a -> 'b) 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
```

Out[75]:

```
type _ typ =
    TInt : int typ
  | TBool : bool typ
  | TLam : 'a typ * 'b typ -> ('a -> 'b) typ
```

Out[75]:

```
type 'a value =
    VInt : int -> int value
  | VBool : bool -> bool value
  | VLam : ('a value -> 'b value) -> ('a -> 'b) value
and 'a expr =
    Var : string * 'a typ -> 'a expr
  | App : ('a -> 'b) expr * 'a expr -> 'b expr
  | Lam : string * 'a typ * 'b expr -> ('a -> 'b) 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
```

In [76]:

```
let rec typ_eq : type a b. a typ -> b typ -> (a, b) eql option =
  fun t1 t2 -> match t1, t2 with
  | TInt, TInt -> Some Refl
  | TBool, TBool -> Some Refl
  | TLam (t1,t2), TLam (u1,u2) ->
    begin match typ_eq t1 u1, typ_eq t2 u2 with
    | Some Refl, Some Refl -> Some Refl
    | _ -> None
    end
  | _ -> None
```

Out[76]:

```
val typ_eq : 'a typ -> 'b typ -> ('a, 'b) eql option = <fun
>
```

In [77]:

```
type env =
  | Empty : env
  | Extend : string * 'a typ * 'a value * env -> env
```

Out[77]:

```
type env = Empty : env | Extend : string * 'a typ * 'a valu
e * env -> env
```

Evaluation

```
eval : a expr -> a value
```

- Limitations:
  - Evaluation is defined only on closed terms.
  - Type checking for variables use done at runtime.

In [78]:

```
let rec eval : type a. env -> a expr -> a value =
  fun env e -> match e with
  | Var (s,t) ->
     begin match env with
       | Empty -> failwith "not a closed term"
       | Extend (s',t',v,env') ->
           if s = s' then
             match typ_eq t t' with
             | Some Refl -> v
             | None -> failwith "types don't match"
           else eval env' e
     end
  | App (e1,e2) ->
    let VLam f = eval env e1 in
    f (eval env e2)
  | Lam (s,t,e) ->
      VLam (fun v -> eval (Extend (s, t, v, env)) e)
  | Val v -> v
  | Plus (e1, e2) ->
    let VInt i1, VInt i2 = eval env e1, eval env e2 in
    VInt (i1 + i2)
  | Mult (e1, e2) ->
    let VInt i1, VInt i2 = eval env e1, eval env e2 in
    VInt (i1 * i2)
  | Ite (p,e1,e2) ->
    let VBool b = eval env p in
    if b then eval env e1 else eval env e2

let eval e = eval Empty e
```

Out[78]:

```
val eval : env -> 'a expr -> 'a value = <fun>
```

Out[78]:

```
val eval : 'a expr -> 'a value = <fun>
```

Typing catches errors

In [79]:

```
eval @@ App (Lam ("x",TInt,Val(VInt 0)), (Val (VInt 10)))
```

Out[79]:

```
- : int value = VInt 0
```

In [80]:

```
eval @@ App (Lam ("x",TInt,Val(VInt 0)), (Val (VBool 10)))
```

```
File "[80]", line 1, characters 46-56:
Error: This expression has type bool value
       but an expression was expected of type int value
       Type bool is not compatible with type int
   1: eval @@ App (Lam ("x",TInt,Val(VInt 0)), (Val (VBool
 10)))
```

Runtime errors

In [81]:

```
eval @@ App (Lam ("x",TBool, Plus (Val (VInt 10), Var ("x", TInt))), Val
```

```
Exception: Failure "types don't match".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "[78]", line 20, characters 40-51
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

In [82]:

```
eval @@ Var ("x", TInt)
```

```
Exception: Failure "not a closed term".
Raised at file "stdlib.ml", line 33, characters 22-33
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

Fixing the runtime errors

- Use the host language (OCaml) features for encoding the features in the object language (lambda calculus).
  - The feature we use here is abstraction.
  - Use `fun (x : TInt) -> e` to encode `Lam "x" TInt e`

In [83]:

```
type 'a value =
  | VInt : int -> int value
  | VBool : bool -> bool value
  | VLam : ('a expr -> 'b expr) -> ('a -> 'b) value

and 'a expr =
  | App : ('a -> 'b) expr * 'a expr -> 'b 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
```

Out[83]:

```
type 'a value =
    VInt : int -> int value
  | VBool : bool -> bool value
  | VLam : ('a expr -> 'b expr) -> ('a -> 'b) value
and 'a expr =
    App : ('a -> 'b) expr * 'a expr -> 'b 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
```

Higher Order Abstract Syntax

- This technique is known as Higher-Order Abstract Syntax (HOAS).
- Downside is that we can no longer **look inside** the abstraction.
  - In the term `Lam f`, we cannot do anything on `f` except apply it.
  - Cannot implement full-beta reduction for example
  - or do code transformation inside the body of lambda.

Evaluator

In [84]:

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

Out[84]:

```
val eval : 'a expr -> 'a value = <fun>
```

Static errors remain static

In [85]:

```
eval @@ App (Val (VLam (fun x -> Val(VInt 0))), (Val (VInt 10)))
```

Out[85]:

```
- : int value = VInt 0
```

In [86]:

```
eval @@ App (Val (VLam (fun x -> Val(VInt 0))), (Val (VBool 10)))
```

```
File "[86]", line 1, characters 60-62:
Error: This expression has type int but an expression was e
xpected of type
         bool
   1: eval @@ App (Val (VLam (fun x -> Val(VInt 0))), (Val
 (VBool 10)))
```

Dynamic errors also become static

In [87]:

```
eval @@ App (Val (VLam (fun (x : bool expr) -> Plus (Val (VInt 10), x)))
```

```
File "[87]", line 1, characters 68-69:
Error: This expression has type bool expr
       but an expression was expected of type int expr
       Type bool is not compatible with type int
   1: eval @@ App (Val (VLam (fun (x : bool expr) -> Plus
 (Val (VInt 10), x))), Val (VBool true))
```