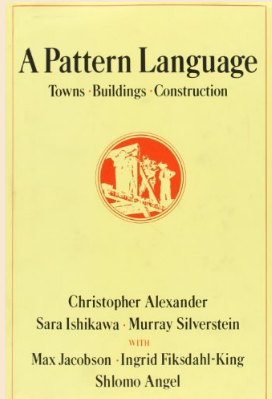
Generalized algebraic data types

February 2018

This time: GADT programming patterns



Recapitulation: philosophical

Phantom types protect **abstractions** against misuse.
GADTs also protect **definitions**.

GADTs lead to rich types which can be **viewed as propositions**.

```
type ('a, 'b, 'c) max    (* max(a,b) = c *)
```

Descriptive data types lead to useful **function types**.

```
val ? : ('a,'n) gtree → ('a,'n) gtree
```

GADT type indexes vary across constructors.

```
type _ t = Int  : int t
         | Bool : bool t
```

We have **families** of types: a type per nat, per tree depth, etc.

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

GADTs are about **type equalities** (and sometimes **inequalities**).

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

Type equalities are **revealed by examining data**.

```
let cast : type a b. (a, b) eql -> a -> b =
  fun Refl x -> x
```

Compilers use the richer types to **generate better code**.

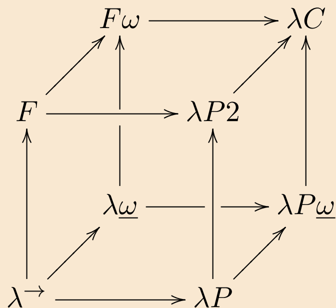
```
(function p (field 1 p))
```

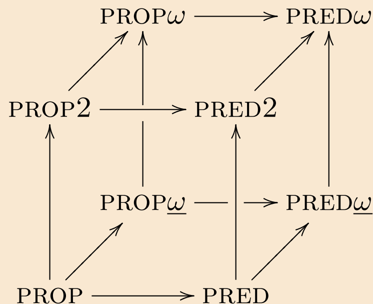
Pattern: Singleton types

Without dependent types we can't write predicates about data.

Using one type per value allows us to simulate value indexing.

Singleton types: Lambda and logic cubes





Singleton sets bring propositional logic closer to predicate logic.

$$\forall A.B \qquad \forall\{x\}.B \qquad \forall x \in A.B$$

Singletons: a **1-to-1 correspondence** between types and data

```
type z = Z
type _ s = S : 'n → 'n s
```

Then **predicates** about types stand in for predicates involving data:

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

This leads to a **relational style** of programming:

```
type (_,_,_) add =
    AddZ : (z,'n,'n) add
  | AddS : ('m,'n,'o) add → ('m s,'n,'o s) add
```

Pattern: Building GADT values

It's not always possible to determine index types statically.

For example, the depth of a tree might depend on user input.

How might a function `make_t` build a value of a GADT type `_ t`?

```
let make_t : type a.string → a t = ...
```

Building GADT values: two approaches

How might a function `make_t` build a value of a GADT type `_ t`?

```
let make_t : type a.string → a t = ✗
```

Building GADT values: two approaches

How might a function `make_t` build a value of a GADT type `_ t`?

```
let make_t : type a.string → a t = ✗
```

With **existentials** `make_t` **builds** a value of type `'a t` for **some** `'a`.

Building GADT values: two approaches

How might a function `make_t` build a value of a GADT type `_ t`?

```
let make_t : type a.string → a t = ✗
```

With **existentials** `make_t` **builds** a value of type `'a t` for **some** `'a`.

With **universals** the caller of `make_t` must **accept** `'a t` for **any** `'a`.

Building GADT values with existentials

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

Define a type of **trees whose depth is not exposed**:

```
type 'a egtree = E : ('a,'n) gtree -> 'a egtree
```

`buildegE` builds a tree whose **depth does not appear in the type**:

```
let rec buildegE : 'a. int -> 'a -> 'a egtree =
  fun depth v -> match depth with
      0 -> E EmptyG
    | n -> let E child = buildegE (n - 1) v in
           E (TreeG (child, v, child))
```

Building GADT values with universals

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

Define a type of **polymorphic continuations** that accept trees:

```
type ('a, 'r) agtree =
  { ag: 'n. ('a,'n) gtree -> 'r }
```

`buildegA` takes a continuation that **accepts trees of any depth**:

```
let rec buildegA : 'a 'r.int → 'a → ('a,'r) agtree → 'r =
  fun depth v { ag = return } → match depth with
      0 → return EmptyG
    | n → buildegA (n - 1) v
           { ag = fun child →
                   return (TreeG (child, v, child)) }
```

Pattern: Building evidence

With type refinement we learn about types by inspecting values.

Predicates return useful *evidence* rather than `true` or `false`.

Predicates without evidence

```
let is_empty : 'a .'a tree → bool =
  function
    Empty → true
  | Tree _ → false
```

```
if not (is_empty t) then      (* t : a tree *)
  top t                       (* t : a tree *)
else
  None                        (* t : a tree *)
```

```
type _ is_zero =
  Is_zero : z is_zero
| Is_succ : _ s is_zero
```

```
let is_empty : type a n.(a,n) dtree → n is_zero =
  function
    EmptyD → Is_zero
  | TreeD _ → Is_succ
```

```
match is_empty t with         (* t : (a, n) dtree   *)
  Is_succ → Some (topD t)  (* t : (a, m s) dtree *)
| Is_zero → None              (* t : (a, z) dtree   *)
```

Summary

Singleton types

Building GADT values

Building evidence