Generic programming

March 2018

Last time: inductive families

```
Vec : Set → ℕ → Set
```

This time: generic programming

```
val show : 'a → string
```

Generic functions

unit

```
type unit =
 Unit : unit
```

booleans

```
type bool =
   False: bool
 | True : bool
```

natural numbers

```
type nat =
   Zero: nat
 | Succ: nat → nat
```

sums

```
type ('a,'b) sum =
  Left : 'a → ('a,'b) sum
| Right: 'b → ('a,'b) sum
```

pairs

```
type ('a,'b) pair =
 Pair: 'a * 'b → ('a,'b) pair
```

lists

```
type 'a list =
  Nil : 'a list
| Cons: 'a * 'a list → 'a list
```

One approach: a signature & implicit functions for each operation

```
module type SHOW =
sig
  type t
  val show : t → string
end

let show : {S:SHOW} → S.t → string =
  fun {S:SHOW} x = S.show x
```

unit

```
implicit module Show_unit = struct
  type t = unit
  let show = function Unit → "Unit"
end
```

pairs

```
implicit module Show_pair {A:SHOW} {B:SHOW} = struct
  type t = (A.t, B.t) pair
  let show = function
    Pair (x y) → "(Pair "^ A.show x ^","^ B.show y ^")"
end
```

lists

```
implicit module Show_list {A:SHOW} = struct
  type t = A.t list
  let rec show : t → string = function
      Nil → "Nil"
    | Cons (x,xs) → "(Cons "^ A.show x ^","^ show xs ^")"
end
```

Operations defined on (most) data

equality

`{E:EQ} → E.t → E.t → bool`

hashing

`{H:HASH} → H.t → int`

ordering

`{O:ORD} → O.t → O.t → int`

mapping

`{F:FUNCTOR} → ('a → 'b) → 'a F.t → 'b F.t`

pretty-printing

`{S:SHOW} → S.t → string`

parsing

`{R:READ} → string → R.t`

serialising

`{B:BIN} → B.t → string`

sizing

`{S:SIZE} → S.t → int`

Generic functions and parametricity

Some built-in OCaml functions violate **parametricity**:

```
val (=) : 'a → 'a → bool
```

With **implicits** we can do better, parameterising by equality:

```
val (=) : {E:EQ} → E.t → E.t → bool
```

This is still inconvenient (one instance per **type/operation pair**).
Better: parameterise by **data shape**:

```
val (=) : {D:DATA} → D.t → D.t → bool
```

Aim: one instance per type + one implementation of each function.

Data shape descriptions: type-indexed values

Idea: an instance of some signature T for each OCaml type:

```
implicit module T_int : T with type t = int
implicit module T_bool : T with type t = bool
implicit module T_pair{A:T} {B:T} :
   T with type t = A.t * B.t
implicit module T_list{A:T} : T with type t = A.t list
implicit module T_option{A:T} : T with type t = A.t option
(* etc. *)
```

`int` is represented by an implicit module

```
T_int : T with type t = int
```

`int * bool` is represented by an implicit module

```
T_pair{T_int}{T_bool} : T with type t = int * bool
```

`int option list` is represented by an implicit module

```
T_list{T_option{T_int}} : T with type t = int option list
```

(etc.)

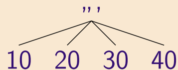
Data as trees

```
L ()
```

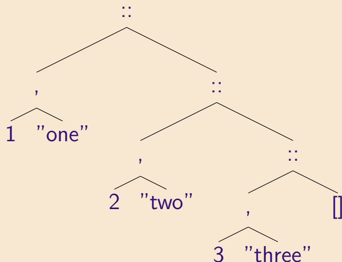
L

()

```
(10,20,30,40)
```

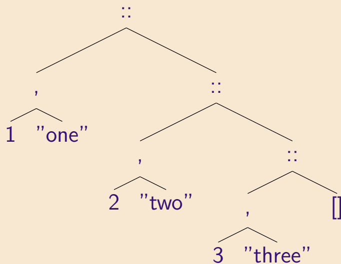



```
[(1, "one"); (2, "two"); (3, "three")]
```



Generic operations: three questions about data

1. What **type** is this data?
2. What are its **subnodes**?
3. What about the **recursive** case?



1. Examining types

A type **representation**:

```
type _ type_rep
```

A type equality **test** that returns a proof on success:

```
val eqrep : 'a type_rep → 'b type_rep →
  ('a,'b) eql option
```

Comparing type representations:

```
# eqrep ty_int ty_float
- : (int, float) eql option = None

# eqrep ty_int ty_int
- : (int, int) eql option = Some Refl
```

How should we define type_rep?

An initial attempt:

```
type _ type_rep =
    Int : int type_rep
  | Bool : bool type_rep
  | List : 'a type_rep → 'a list type_rep
  | Option : 'a type_rep → 'a option type_rep
  | Pair : 'a type_rep * 'b type_rep → ('a * 'b) type_rep
```

An initial attempt at type equality:

```
let rec eqrep :
 type a b.a typerep → b typerep → (a,b) eql option =
 fun l r → match l, r with
    Int, Int → Some Refl
  | Bool, Bool → Some Refl
  | List s, List t → (match eqrep s t with
                       | Some Refl → Some Refl
                       | None → None)
  | Option s, Option t → (match eqrep s t with
                           | Some Refl → Some Refl
                           | None → None)
  | Pair (s1, s2), Pair (t1, t2) →
     (match eqrep s1 t1, eqrep s2 t2 with
      | Some Refl, Some Refl → Some Refl
      | _ → None)
  | _ → None
```

An initial attempt at type equality:

```
let rec eqrep :
 type a b.a typerep → b typerep → (a,b) eql option =
 fun l r → match l, r with
    Int, Int → Some Refl
  | Bool, Bool → Some Refl
  | List s, List t → (match eqrep s t with
                         | Some Refl → Some Refl
                         | None → None)
  | Option s, Option t → (match eqrep s t with
                             | Some Refl → Some Refl
                             | None → None)
  | Pair (s1, s2), Pair (t1, t2) →
     (match eqrep s1 t1, eqrep s2 t2 with
      | Some Refl, Some Refl → Some Refl
      | _ → None)
  | _ → None
```

Problem: this representation has no support for user-defined types.

Defining

```
type 'a t = ..
```

Extending

```
type 'a t +=
  A : int list → int t
| B : float list → 'a t
```

Constructing

```
A [1;2;3] (* No different to standard variants *)
```

Matching

```
let f : type a. a t → string = function
  A _ → "A"
| B _ → "B"
| _ → "?"    (* All matches must be open *)
```

Types & type equality, extensibly (interface)

A single **extensible variant** for type representations:

```
type _ type_rep = ..
```

A **signature** with a representation and equality check:

```
module type TYPEABLE = sig
  type t
  val type_rep : t type_rep
  val eqrep : 'other type_rep → (t,'other) eql option
end
```

A **top-level function** for determining type equality:

```
let eqty: {S:TYPEABLE} → {T:TYPEABLE} → (S.t, T.t) eql
  = fun {S:TYPEABLE} {T:TYPEABLE} → S.eqty T.type_rep
```

Types & type equality, extensibly (implementation)

A type_rep **constructor** for each type:

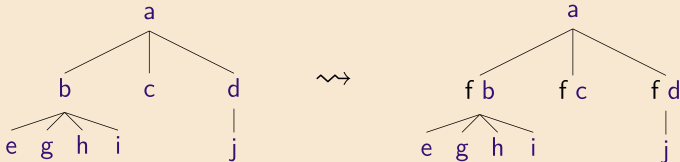
```
type _ type_rep += List : 'a type_rep → 'a list type_rep
```

A TYPEABLE **instance** for each type:

```
implicit module Typeable_list{A:TYPEABLE} = struct
 type t = A.t list
 let type_rep = List A.type_rep
 let eqrep: type b.b type_rep → (A.t list,b) eql option
  = function | List b → (match A.eqrep b with
                          | Some Refl → Some Refl
                          | None → None)
             | _ → None
end
```

2. Accessing subnodes

gmapT



gmapQ

An interface for accessing subnodes

A second type-indexed value for **describing data**:

```
module type DATA
```

Generic functions: **t**raversals and **q**ueries:

```
type genericT = {D:DATA} → D.t → D.t
type 'u genericQ = {D:DATA} → D.t → 'u

val gmapT : genericT → genericT
val gmapQ : 'u genericQ → 'u list genericQ
```

A signature for accessing subnodes

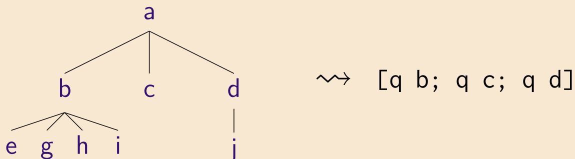
The DATA signature provides **access to subnodes**:

```
module type DATA = sig
 type t
 module Typeable : TYPEABLE with type t = t
 val gmapT : genericT → t → t
 val gmapQ : 'u genericQ → t → 'u list
end
```

An instance of DATA for each type:

```
implicit module Data_int :
   DATA with type t = int
implicit module Data_list{A:DATA} :
   DATA with type t = A.t list
(etc.)
```

Polymorphic types for generic queries: gmapQ



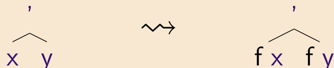

```
type 'u genericQ = {D:DATA} → D.t → 'u

val gmapQ : 'u genericQ → 'u list genericQ
```

Traversing datatypes: primitive types

$$\texttt{x} \rightsquigarrow \texttt{x}$$

`gmapT {Data_int} f`

```
implicit module Data_int : DATA with type t = int =
struct
  type t = int
  module Typeable = Typeable_int
  let gmapT f x = x
  let gmapQ f x = []
end
```

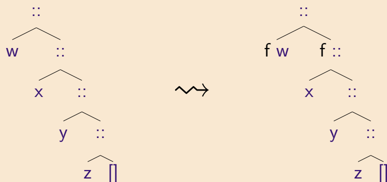



```
gmapT {Data_pair{A}{B}} f
```

With DATA instances A and B for the type parameters:

```
implicit module Data_pair {A: DATA} {B: DATA}
 : DATA with type t = A.t * B.t =
struct
  type t = A.t * B.t
  module Typeable =
    Typeable_pair{A.Typeable}{B.Typeable}
  let gmapT f (x, y) = (f x, f y)
  let gmapQ q (x, y) = [q x; q y]
end
```

Traversing datatypes: lists



gmapT (list a) f

```
implicit module rec Data_list {A: DATA} :
   DATA with type t = A.t list = struct
 (* ... *)
 let gmapT f l =
   match l with [] → [] | x :: xs → f x :: f xs
 (* ... *)
end
```

(Disclaimer: `implicit module rec` not yet supported)

3. Handling recursion

`everywhere`: apply `f` to subnodes, then to the node itself

```
let rec everywhere : genericT → genericT =
  fun f {X:DATA} x → f (gmapT (everywhere f) x)
```

In practice a few more annotations are needed:

```
let rec everywhere : genericT → genericT =
 fun (f : genericT) {X:DATA} x →
    f ((gmapT (everywhere f) : genericT) x)
```

Generic maps, top down

`everywhere'`: apply `f` to the node itself, then to subnodes

```
let rec everywhere' : genericT → genericT =
  fun f {X:DATA} x → gmapT (everywhere' f) (f x)
```

With annotations:

```
let rec everywhere' : genericT → genericT =
  fun (f : genericT) {X:DATA} x →
     (gmapT (everywhere' f) : genericT) (f {X} x)
```

Using generic maps

`everywhere` builds a generic traversal from a generic traversal

```
val everywhere : genericT → genericT
```

`mkT` builds a generic traversal from a monomorphic function:

```
let mkT: {T:TYPEABLE} → (T.t → T.t) → genericT =
  fun {T:TYPEABLE} g {D: DATA} x →
    match eqty {T} {D:TYPEABLE} with
      | Some Refl → g x
      | None → x
```

Using `everywhere` with `mkT`:

```
everywhere (mkT succ)
  [(false, 1); (false, 2); (true, 3)]
```

Generic queries

`everything f g` applies `g` to every node & combines results with `f`:

```
let rec everything join g {X: DATA} x =
  fold_left join (g x) (gmapQ (everything join g) x)
```

Using generic queries

`everything` builds a generic query from a generic query

```
val everything :
  ('r → 'r → 'r) → 'r genericQ → 'r genericQ =
```

`mkQ` builds a generic query from a monomorphic function:

```
let mkQ: 'u.{T:TYPEABLE} → 'u → (T.t → 'u) → 'u genericQ =
  fun {T:TYPEABLE} u g {D: DATA} x →
   match eqty {T} {D.Typeable} with
      | Some Refl → g x
      | None → u
```

Using `everything` with `mkQ`:

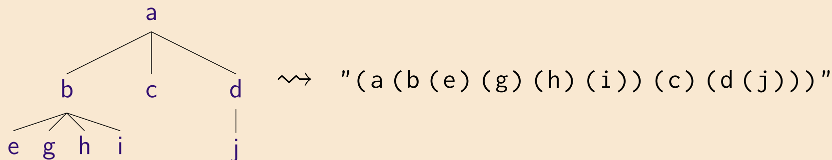
```
everything (@) (mkQ [] (fun x → [x]))
  [(1,false); (2,true)]
```

Generic printing

Add an additional field to `DATA` for distinguishing constructors:

```
module type DATA = sig
  type t
  module Typeable : TYPEABLE with type t = t
  val gmapT : genericT → t → t
  val gmapQ : 'u genericQ → t → 'u list
  val constructor_: t → string
end
```

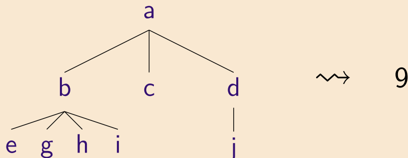
A generic printing function




```
let rec gshow {D:DATA} (v : D.t) =
  "("^ constructor_ v
     ^ String.concat " " (gmapQ gshow v) ^ ")"
```

Generic sizing

Computing value size generically



$\rightsquigarrow$ 9

```
let sum l = List.fold_left (+) 0 l

let rec gsize {D:DATA} (v : D.t) =
  1 + sum (gmapQ gsize v)


gsize 3

gsize [1;2;3]

gsize [(1,false); (2,false); (3,true)]
```

Main remaining problem: performance

Generic traversals are slow!

Solution: **staging** (next week)

`.< e >.`

Generic programming in Agda

(a brief glimpse)

Generic programming with dependent types: overview

Define a **universe** of
  codes (data describing types)
  interpretation (large eliminations for codes)

Define generic functions by **induction on codes**

Recap: algebraic data types as sums-of-products

introduce recursive type

type parameter

```
type 'a list =
   Nil : 'a list
 | Cons : 'a * 'a list -> 'a list
```

binary sum

recursive occurrence

$$\lambda\alpha.\mu\ell.1 + (\alpha \times \ell)$$

Codes for sums-of-products[1]

```
data Konst : Set where
  bool : Konst
  nat  : Konst

data Atom : Set where
  K : Konst → Atom
  I :  Atom -- (the recursive variable)

Prod Sum : Set
Prod = List Atom
Sum  = List Prod
```

[1](from *Type-directed diffing of structured data*, Miraldo et al, 2017)

Interpreting codes

$$
\begin{aligned}
&[\![_]\!]_{\mathsf{k}} : \mathsf{Konst} \to \mathsf{Set}\\
&[\![\ \mathsf{bool}\ ]\!]_{\mathsf{k}} = \mathsf{Bool}\\
&[\![\ \mathsf{nat}\ ]\!]_{\mathsf{k}} = \mathsf{Nat}
\end{aligned}
$$

$$
\begin{aligned}
&[\![_]\!]_{\mathsf{a}} : \mathsf{Atom} \to (\mathsf{Set} \to \mathsf{Set})\\
&[\![\ \mathsf{K}\ k\ ]\!]_{\mathsf{a}}\ _ = [\![\ k\ ]\!]_{\mathsf{k}}\\
&[\![\ \mathsf{I}\ ]\!]_{\mathsf{a}}\ X = X
\end{aligned}
$$

$$
\begin{aligned}
&[\![_]\!]_{\mathsf{p}} : \mathsf{Prod} \to (\mathsf{Set} \to \mathsf{Set})\\
&[\![\ []\ ]\!]_{\mathsf{p}}\ X = \top\\
&[\![\ \alpha :: p\ ]\!]_{\mathsf{p}}\ X = [\![\ \alpha\ ]\!]_{\mathsf{a}}\ X \times [\![\ p\ ]\!]_{\mathsf{p}}\ X
\end{aligned}
$$

$$
\begin{aligned}
&[\![_]\!]_{\mathsf{s}} : \mathsf{Sum} \to (\mathsf{Set} \to \mathsf{Set})\\
&[\![\ []\ ]\!]_{\mathsf{s}}\ X = \bot\\
&[\![\ \pi :: s\ ]\!]_{\mathsf{s}}\ X = [\![\ \pi\ ]\!]_{\mathsf{p}}\ X \uplus [\![\ s\ ]\!]_{\mathsf{s}}\ X
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{Fix}\ (\sigma : \mathsf{Sum}) : \mathsf{Set}\ \mathsf{where}\\
&\quad \langle_\rangle : [\![\ \sigma\ ]\!]_{\mathsf{s}}\ (\mathsf{Fix}\ \sigma) \to \mathsf{Fix}\ \sigma
\end{aligned}
$$

Define equality by **induction on codes**

```
eqk : (κ : Konst) → ⟦ κ ⟧k → ⟦ κ ⟧k → Bool
eqk bool x y with x Data.Bool.≟ y
... | yes _ = true
... | no  _ = false
eqk nat x y with x Data.Nat.≟ y
... | yes _ = true
... | no  _ = false


eqp : (π : Prod) → (σ : Sum) → ⟦ π ⟧p (Fix σ) → ⟦ π ⟧p (Fix σ) → Bool
eqp [] σ x y = true
eqp (α :: π) σ (u , v) (w , x) = eqa α σ u w ∧ eqp π σ v x
```

(etc.)

Equality for lists:

```
listF : Sum
listF = [] :: (K nat :: I :: []) :: []
```

```
list = ⟦ listF ⟧s (Fix listF)
```

```
eqList : list → list → Bool
eqList = eqs listF listF
```

Testing by type checking:

```
check1 : eqList
  (cons zero (cons (suc zero) nil))
  (cons zero (cons (suc (suc zero)) nil)) ≡ false
check1 = refl
```

Next time: staging

.< e >.