Inductive families

February 2018

This time: Inductive families

```
Vec : Set → ℕ → Set
```

GADTs: difficulties

Simple types recap

$$0$$

$$1$$

$$A \times B$$

$$A + B$$

Simple structure, so lots of "models"
(sets, relations, propositions, semirings)

GADTs are about **type equalities**.

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

GADTs reveal things about **types** when you examine **data**.

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

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

```
val max : ('a,'b,'c) max → 'a → 'b → 'c
```

GADT problems: existentials (or universals)

With GADTs, adding indexing requires existentials.

An **existential** type that **hides** the depth index:

```
type 'a edtree =  E : ('a, _) dtree → 'a edtree
```

Constructing a **depth-indexed** tree from an **unindexed** tree:

```
let rec dify : 'a. 'a tree -> 'a edtree = function
    Empty -> E EmptyD
  | Tree (l, x, r) ->
    let E l' = dify l in
    let E r' = dify r in
    E (TreeD (l', x, r', max_depth l' r'))
```

There's no way to **relate** the depth index to the unindexed input.

GADT problems: singletons

With GADTs we need a **shadow world of types** for index values:

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

# let zero = Z;;
val zero : z = Z
# let three = S (S (S Z));;
val three : z s s s = S (S (S Z))
```

GADT problems: relational programming

With GADTs we can talk about arguments, but not about results

What we **mean**:

$$max(m, n) \equiv 0$$

What we **say**:

```
('m, 'n, 'o) max
```

GADT problems: untyped indexes

With GADTs nothing prevents constructing meaningless types.

An **inhabited** type:

```
(* max(1,0) ≡ 1 *)
(z s, z, z s) max
```

An **uninhabited** type:

```
(* max(1,2) ≡ 0 *)
(z s, z s s, z) max
```

A **meaningless** type:

```
(* max(int,string) ≡ float *)
(int, string, float) max
```

With **inductive families** (and **dependent types** generally) things become **much simpler**…

Agda primer

Dependent functions and abbreviations

The dependent function space (Π) is written like this

$$(x : A) \to B$$

Implicit arguments:

$$\forall \{x : A\} \to B$$

For non-dependent functions (x not used in B), abbreviate:

$$A \to B$$

If A can be inferred, abbreviate:

$$\forall\ x \to B$$

Simple data:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Nat}\ :\ \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{zero}\ :\ \mathsf{Nat} \\
\quad \mathsf{suc}\ \ :\ \mathsf{Nat} \to \mathsf{Nat}
\end{array}
$$

Parameterised data:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{Tree}\ (\alpha\ :\ \mathsf{Set})\ :\ \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{Empty}\ \ :\ \mathsf{Tree}\ \alpha \\
\quad \mathsf{Branch}\ :\ \mathsf{Tree}\ \alpha \to \alpha \to \mathsf{Tree}\ \alpha \to \mathsf{Tree}\ \alpha
\end{array}
$$

Indexed data:

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{DTree}\ (\alpha\ :\ \mathsf{Set})\ :\ \mathsf{Nat} \to \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{Empty}\ :\ \mathsf{DTree}\ \alpha\ \mathsf{zero} \\
\quad \mathsf{Branch}\ :\ \forall\ \{x\ y\ :\ \mathsf{Nat}\} \to \mathsf{DTree}\ \alpha\ x \to \alpha \to \mathsf{DTree}\ \alpha\ y \to \\
\qquad \mathsf{DTree}\ \alpha\ (\mathsf{suc}\ (\mathsf{max}\ x\ y))
\end{array}
$$

Functions are written in **equational style**:

$$\mathsf{max} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$\mathsf{max}\ \mathsf{zero}\ r = r$$
$$\mathsf{max}\ r\ \mathsf{zero} = r$$
$$\mathsf{max}\ (\mathsf{suc}\ l)\ (\mathsf{suc}\ r) = \mathsf{suc}\ (\mathsf{max}\ l\ r)$$

with clauses can compute with pattern variables on the **left of =**:

$$\mathsf{max}_2 : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$\mathsf{max}_2\ \mathsf{zero}\ r = r$$
$$\mathsf{max}_2\ r\ \mathsf{zero} = r$$
$$\mathsf{max}_2\ (\mathsf{suc}\ l)\ (\mathsf{suc}\ r)\ \mathsf{with}\ \mathsf{max}_2\ l\ r$$
$$\ldots \mid m = \mathsf{suc}\ m$$

Holes

Agda supports **hole-driven development**.
Idea: leave holes in programs; fill interactively with Agda's help.

$$\begin{array}{l} \mathsf{min} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{min}\ \mathsf{zero}\ r = \mathsf{zero} \\ \mathsf{min}\ r\ \mathsf{zero} = \mathsf{zero} \\ \mathsf{min}\ (\mathsf{suc}\ l)\ (\mathsf{suc}\ r) = \{!!\} \end{array}$$

```
Goal: Nat
------------
r : Nat
l : Nat
```

Benefits of rich types:

- Clearly describe intent
- Exclude incorrect programs
- Generate faster code
- Support interactive development (***new!***)

GADTs, improved

Indexing by terms: no more singletons

With **GADTs** we need a shadow world of types:

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

# let zero = Z;;
val zero : z = Z
# let three = S (S (S Z));;
val three : z s s s = S (S (S Z))
```

With **term-indexed types** we can use simple data definitions:

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

Indexing by terms: no more relational programming

With **GADTs** type-level functions are written in relational style:

```
('m, 'n, 'o) max
```

With **term-indexed types** we can use simple functions:

$$
\begin{aligned}
&\mathsf{max} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\
&\mathsf{max}\ \mathsf{zero}\ r = r\\
&\mathsf{max}\ r\ \mathsf{zero} = r\\
&\mathsf{max}\ (\mathsf{suc}\ l)\ (\mathsf{suc}\ r) = \mathsf{suc}\ (\mathsf{max}\ l\ r)
\end{aligned}
$$

Indexing with max:

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{DTree}\ (\alpha : \mathsf{Set}) : \mathsf{Nat} \to \mathsf{Set}\ \mathsf{where}\\
&\quad \mathsf{Empty} : \mathsf{DTree}\ \alpha\ \mathsf{zero}\\
&\quad \mathsf{Branch} : \forall\ \{x\ y : \mathsf{Nat}\} \to \mathsf{DTree}\ \alpha\ x \to \alpha \to \mathsf{DTree}\ \alpha\ y \to\\
&\qquad \mathsf{DTree}\ \alpha\ (\mathsf{suc}\ (\mathsf{max}\ x\ y))
\end{aligned}
$$

Indexing by terms: no more untyped indexes

With **GADTs** we can construct meaningless types (because all indexes are in ★):

```
(* max(int,string) ≡ float *)
(int, string, float) max
```

With inductive families, **indexes are classified**:

$$\mathsf{data}\ \mathsf{Dtree}\ (\alpha : \mathsf{Set}) : \mathsf{Nat} \to \mathsf{Set}\ \mathsf{where}$$

This is well-typed:

$$\mathsf{Dtree}\ \mathsf{Nat}\ (\mathsf{suc}\ \mathsf{zero})$$

...but this is an error:

$$\mathsf{Dtree}\ \mathsf{Nat}\ \mathsf{Bool}$$

Indexing by terms: no more existentials

GADTs require existentials to add indexing:

```
type 'a edtree =  E : ('a, _) dtree → 'a edtree
let rec dify : 'a. 'a tree -> 'a edtree = ...
```

Term-indexed result **types** can mention input **terms**:

$$\mathsf{dify} : \forall\{\alpha\} \to (t : \mathsf{Tree}\ \alpha) \to \mathsf{DTree}\ \alpha\ (\mathsf{depth}\ t)$$
$$\mathsf{dify}\ \mathsf{Empty} = \mathsf{Empty}$$
$$\mathsf{dify}\ (\mathsf{Branch}\ t\ x\ t_1) = \mathsf{Branch}\ (\mathsf{dify}\ t)\ x\ (\mathsf{dify}\ t_1)$$

Beyond GADTs

Internal vs external verification

GADTs support **internal verification** by indexing data:

```
val top : ('a,'n) gtree → 'n
```

But indexing by **every property** is unwieldy and non-modular.
(Consider: how can we define a **sorted** `gtree` using `gtree`?)

Agda's dependent types support **external verification**
— separating data and function definition from properties:

$$\mathsf{max{-}comm} \;:\; \forall\ \{m\ n\} \to (\mathsf{max}\ m\ n) \equiv (\mathsf{max}\ n\ m)$$

$$\mathsf{swiv{-}depth} \;:\; \forall\ \{\alpha\} \to (t \,:\, \mathsf{Tree}\ \alpha) \to \mathsf{depth}\ t \equiv \mathsf{depth}\ (\mathsf{swivel}'\ t)$$

Large eliminations are an alternative way of defining indexed data.

Idea: functions from data to types.

Defining perfect trees as an **inductive family**:

```
data GTree (α : Set) : Nat → Set where
  Empty : GTree α zero
  TreeG : ∀ {n} → GTree α n → α → GTree α n → GTree α (suc n)
```

Defining perfect trees via a **recursive function**:

```
gtree : Set → Nat → Set
gtree α zero = ⊤
gtree α (suc n) = gtree α n × α × gtree α n
```

Example ($n \equiv \mathsf{suc\ zero}$):

$$\begin{aligned}\mathsf{gtree}\ \alpha\ (\mathsf{suc\ zero}) &= \mathsf{gtree}\ \alpha\ \mathsf{zero} \times \alpha \times \mathsf{gtree}\ \alpha\ \mathsf{zero} \\ &= \top \times \alpha \times \top\end{aligned}$$

Large eliminations, continued

Defining perfect trees via a **recursive function**:

$$\mathsf{gtree} : \mathsf{Set} \to \mathsf{Nat} \to \mathsf{Set}$$
$$\mathsf{gtree}\ \alpha\ \mathsf{zero} = \top$$
$$\mathsf{gtree}\ \alpha\ (\mathsf{suc}\ n) = \mathsf{gtree}\ \alpha\ n \times \alpha \times \mathsf{gtree}\ \alpha\ n$$

Program with gtree by **pattern-matching on the index**:

$$\mathsf{swivel} : \forall\ \{\alpha\ n\} \to \mathsf{gtree}\ \alpha\ n \to \mathsf{gtree}\ \alpha\ n$$
$$\mathsf{swivel}\ \{\alpha\}\ \{\mathsf{zero}\}\ t = \mathsf{tt}$$
$$\mathsf{swivel}\ \{\alpha\}\ \{\mathsf{suc}\ n\}\ (l\ ,\ x\ ,\ r) = \mathsf{swivel}\ r\ ,\ (x\ ,\ \mathsf{swivel}\ l)$$

Equality

Exhaustiveness: does a pattern match cover every case?

For **simple data types**: **well-understood, complete**.

(*ML pattern match compilation and partial evaluation*
Sestoft, 1996)

For **GADTs**: **impossible**

(*GADTs and Exhaustiveness: Looking for the Impossible*
Garrigue & Le Normand, 2017)

For **inductive families**: **even harder**

(*Dependent pattern matching and proof-relevant unification*
Cockx, 2017)

Pattern matching & equality

Type equality (for **GADTs**):

First, **expand aliases**. Then types have form $(\mathtt{t}_1,\ \mathtt{t}_2,\ \ldots\ \mathtt{t}_n)\ \mathtt{t}$.

$(\mathtt{t}_1,\ \mathtt{t}_2,\ \ldots\ \mathtt{t}_n)\ \mathtt{t} \equiv (\mathtt{s}_1,\ \mathtt{s}_2,\ \ldots\ \mathtt{s}_n)\ \mathtt{s}$ iff $\mathtt{t}_1 \equiv \mathtt{s}_1 \land \ldots \land \mathtt{t} \equiv \mathtt{s}$

Pattern matching **exposes equalities**, making types (un)equal.

Type equality with **term indexing**:

First, **normalize terms**. Terms equal if normalizations equal (judgemental equality).

Pattern matching exposes equalities, allowing further **computation**.

If we learn $n \equiv \mathsf{zero}$ (propositional equality), can reduce $\mathsf{max}\ n\ n$

Problem: without normalization we can build bogus proofs:

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

OCaml supports **general recursion**:

```
val fix : (('a -> 'b) -> ('a -> 'b)) -> ('a -> 'b)
let rec fix f x = f (fix f) x
```

i.e. (under Curry-Howard correspondence):

$$\forall A \forall B.((A \to B) \to (A \to B)) \to (A \to B)$$

Problem: determining equality involves normalizing terms
(may not terminate, may perform effects)

Next time: generic programming

```
val show : 'a → string
```