Datatypes

Pedro Abreu

Purdue University

Kinds of Datatypes

1. Enumeration Types

Enumeration Types

```
type bool =
  | true
  | false
```

Kinds of Datatypes

1. Enumeration Types (Variants)
2. Recursive Types

Recursive Types

```
type nat =
  | O
  | S of nat
```

Kinds of Datatypes

1. Enumeration Types
2. Recursive Types
3. Mutually Recursive Types

Mutually Recursive Types

```
type 'a even =
| O
| S_even of 'a odd
and 'a odd =
| S_odd  of 'a even
```

Kinds of Datatypes

1. Enumeration Types
2. Recursive Types
3. Mutually Recursive Types
4. Parametrized Types

Parametrized Types

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

Kinds of Datatypes

1. Enumeration Types
2. Recursive Types
3. Mutually Recursive Types
4. Parametrized Types
5. Indexed Types

Indexed Types

```
type _ term =
| Int : int -> int term
| Add : (int -> int -> int) term
| App : ('b -> 'a) term * 'b term -> 'a term
```

Indexed Types

```
type _ term =
| Int : int -> int term
| Add : (int -> int -> int) term
| App : ('b -> 'a) term * 'b term -> 'a term

let rec eval : type a. a term -> a = function
| Int n     -> n
| Add       -> (fun x y -> x+y)
| App(f,x) -> (eval f) (eval x)
```

Impossible Branches

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

let deep : (char t * int) option -> char = function
  | None -> 'c'
```

GADTs goes by many names

- It has been around for a while, but only recently is becoming popular in the fp comunity.
- Type theory (early 90's)
  - Inductive Sets and Families
- Recent Language design
  - Guarded recursive datatypes (Xi et al.)
  - First-class phantom types (Hinze/Cheney)
  - Equality-qualified types (Sheard et al.)
  - Guarded algebraic datatypes (Simonet/Pottier)

ADT Key Properties

- No Confusion

ADT Key Properties

- No Confusion
  - Injectivity

ADT Key Properties

- No Confusion
  - Injectivity
    - For any constructor C, $C\ x = C\ y \rightarrow x = y$

ADT Key Properties

- No Confusion
  - Injectivity
  - Conflict

ADT Key Properties

- No Confusion
  - Injectivity
  - Conflict
    - $C_1 \neq C_2$
    - e.g. $nil \neq cons$

ADT Key Properties

- No Confusion
  - Injectivity
  - Conflict
- Induction (And case analysis)

ADT Key Properties

- No Confusion
  - Injectivity
  - Conflict
- Induction (And case analysis)
- Aciclicity

ADT Key Properties

- No Confusion
  - Injectivity
  - Conflict
- Induction (And case analysis)
- Aciclicity
  - No term is smaller than itself
  - Important for cycle detection during unification

How does OCaml GADTs relate to Coq Inductives?

How does OCaml GADTs relate to Coq Inductives?

Dependent Types

Coq Inductive

```
Inductive ilist {A: Set} : nat → Set :=
| Nil : ilist O
| Cons : forall n, A → ilist n → ilist (S n).
```

Coq Impossible Branches

```
Inductive ilist {A: Set} : nat → Set :=
| Nil : ilist O
| Cons : forall n, A → ilist n → ilist (S n).

Definition hd n (ls: ilist (S n)): A :=
match ls with
| Cons h hs ⇒ h
end.
```

Coq Impossible Branches

```
Inductive ilist {A: Set} : nat → Set :=
| Nil : ilist O
| Cons : forall n, A → ilist n → ilist (S n).

Definition hd n (ls: ilist (S n)): A :=
match ls with
| Cons h hs ⇒ h
end.
```

Error: Non exhaustive pattern-matching: no clause found for pattern Nil

Coq Impossible Branches I

```
Inductive ilist {A: Set} : nat → Set :=
| Nil : ilist O
| Cons : forall n, A → ilist n → ilist (S n).

Definition hd n (ls: ilist (S n)): A :=
match ls in (ilist n') return (n' = S n) → A with
| Nil ⇒ fun eq ⇒
    False_rect A (neq_succ_0 n (eq_sym eq))
| Cons h hs ⇒ fun _ ⇒ h
end eq_refl.
```

Coq Impossible Branches II

```
Inductive ilist {A: Set} : nat → Set :=
| Nil : ilist O
| Cons : forall n, A → ilist n → ilist (S n).

Definition hd' n (ls: ilist (S n)): A :=
  match ls in (ilist n') return (match n' with
                                  | O ⇒ unit
                                  | S n'' ⇒ A
                                  end) with
    | Nil ⇒ tt
    | Cons h hs ⇒ h
  end.
```

Coq Impossible Branches III

```
Inductive unit_or_double_unit : Type → Type :=
| Unit : unit_or_double_unit unit
| Double_unit : unit_or_double_unit (unit * unit).

Definition twelve (x: unit_or_double_unit unit) : nat :=
  match x in (unit_or_double_unit T) return (T = unit → nat) with
  | Unit ⇒ fun _ ⇒ 12
  | Double_unit ⇒ fun (neq: unit * unit = unit) ⇒
                  (* Proof of False *)
  end eq_refl.
```

Coq Impossible Branches IV

```
Inductive unit_or_double_unit : Type → Type :=
| Unit : unit_or_double_unit unit
| Double_unit : unit_or_double_unit (unit * unit).

Definition twelve (x: unit_or_double_unit unit) : nat :=
match x in (unit_or_double_unit T) return (match T with ????) with
  | Unit ⇒ 12
  | Double_unit ⇒ tt
  end .
```

Coq Positive Checker

```
Inductive Foo : Type → Type :=
| foo : Foo Bar
with
Bar := bar.
```

Coq Positive Checker

```
Inductive Foo : Type → Type :=
| foo : Foo Bar
with
Bar := bar.
```

Error: Non strictly positive occurrence of "Bar" in "Foo Bar".

Coq Positive Checker Workaround

```
Inductive PreFoo : Type :=
| foo : PreFoo.

Inductive Bar : Type := b.

Fixpoint FooWf (f : PreFoo) (t : Type) : Prop :=
  match f with
  | foo => (t = Bar)
  end.

Definition Foo (t : Type) := {f : PreFoo | FooWf f t}.
```

Eliminators

```
Inductive ilist {A: Set} : nat → Set :=
| Nil : ilist O
| Cons : forall n, A → ilist n → ilist (S n).

ilist_ind: forall P : forall n : nat, ilist n → Prop,
       P 0 Nil →
       (forall (n : nat) (a : A) (i : ilist n),
        P n i → P (S n) (Cons n a i)) →
       forall (n : nat) (i : ilist n), P n i
```

Questions?