Cool Inductive Constructions

David Hamelin

Séminaire non-permanent LMF

Plan

1. Variants
2. Simple inductive
3. Mutually recursive inductive
4. Type-dependent inductive
5. Type-dependent inductive for AST
6. Term-dependent inductive
7. Term-dependent inductive for AST

Variant in Coq

```
Variant day : Type :=
| sunday : day
| monday (remote: bool) : day
| tuesday (remote: bool) : day
| wednesday (remote: bool) : day
| thursday (remote: bool) : day
| friday (remote: bool) : day
| saturday.
```

Variant will not generate induction principles.

```
Definition is_remote
  (d: day) : bool :=
  match d with
  | sunday => false
  | monday r => r
  | tuesday r => r
  | wednesday r => r
  | thursday r => r
  | friday r => r
  | saturday => true
  end.
```

Usual inductive types

Abstract Syntax Trees are typically modelled using inductive types.

```
Variant value : Type :=
| vnat (x: nat) : value
| vbool (b: bool) : value.
```

```
Inductive expr : Type :=
| cst (v: value) : expr
| add (x y: expr) : expr
| equal (x y: expr) : expr
| ite (cond tthen eelse: expr)
  : expr.
```

$$V := \mathbb{N} \mid \mathbb{B}$$

$$E := V \mid E + E \mid E = E \mid E \mathbin{?} E : E$$

Evaluating

```
Fixpoint eval (e: expr) : option value :=
match e with
| cst c => Some c
| add x y =>
    match (eval x, eval y) with
    | (Some (vnat x), Some (vnat y)) => Some (vnat (x+y))
    | _ => None
    end
| equal x y =>
    match (eval x, eval y) with
    | (Some x, Some y) => Some (vbool (x == y)) (* == decidable equality on values *)
    | _ => None
    end
| ite cond tthen eelse =>
    match eval cond with
    | (Some (vbool true)) => eval tthen
    | (Some (vbool false)) => eval eelse
    | _ => None
    end
 end.
```

Mutually recursive inductive

Two inductive types can be defined simultaneously :

```
Inductive expr :=
| cst (v:value)
| add (x y : expr)
| equal (x y : expr)
| ite (cond tthen eelse: expr)
| switch (e: expr) (p:
patterns)
```

```
with patterns :=
| default (e: expr)
| ccases (cond: value)
         (tthen: expr)
         (eelse: patterns).
```

Evaluating patterns

```
Fixpoint eval
(e: expr) : option value :=
match e with
| cst c => Some c
| add x y => [...]
| equal x y => [...]
| ite cond tthen eelse => [...]
    end
 | switch e p =>
     match eval e with
     | (Some v) => eval_patterns v p
     | _ => None
     end
 end
```

```
with eval_patterns (x: value)
(p: patterns) : option value :=
  match p with
  | default e => eval e
  | ccases y e p' =>
      if x == y then
        eval e
      else
        eval_patterns x p'
  end.
```

Type-dependent inductive type

```
Inductive list : Type -> Type :=
| nil (T:Type) : list T
| cons (T:Type) (x:T) (xs: list T) : list T
```

```
Definition x : list nat := [1;2;3]
Definition x : list nat := cons nat 1 (cons nat 2 (cons nat 3 (nil nat)))
```

```
Fixpoint sum (l: list nat) : nat :=
  match l with
  | nil _ => 0
  | cons _ x xs => x + (sum xs)
  end.
```

```
Fixpoint map
(A B : Type) (l: list A)
(f: A -> B) : list B :=
  match l with
  | nil A => nil B
  | cons _ x xs =>
      cons B (f x) (map A B xs f)
  end.
```

Type-dependent inductive type for AST

We can use the information carried by the Inductive type to our advantage:

```
Inductive expr : Type -> Type :=
| vnat (x: nat) : expr nat
| vbool (b: bool) : expr bool
| add (x y : expr nat) : expr nat
| equal (x y : expr nat) : expr bool
| ite (cond : expr bool) (T:Type) (tthen
eelse : expr T) : expr T.
```

```
Fixpoint eval (T:Type) (e:expr T) : T :=
match e with
| vnat x => x
| vbool b => b
| add x y => (eval nat x) + (eval nat y)
| equal x y =>(eval nat x) == (eval nat y)
| ite cond T tthen eelse =>
    if eval bool cond then
      eval T tthen
    else
      eval T eelse
end.
```

We don’t need to check for errors anymore, because the typing will enforce the rules.

Term-dependent inductive type

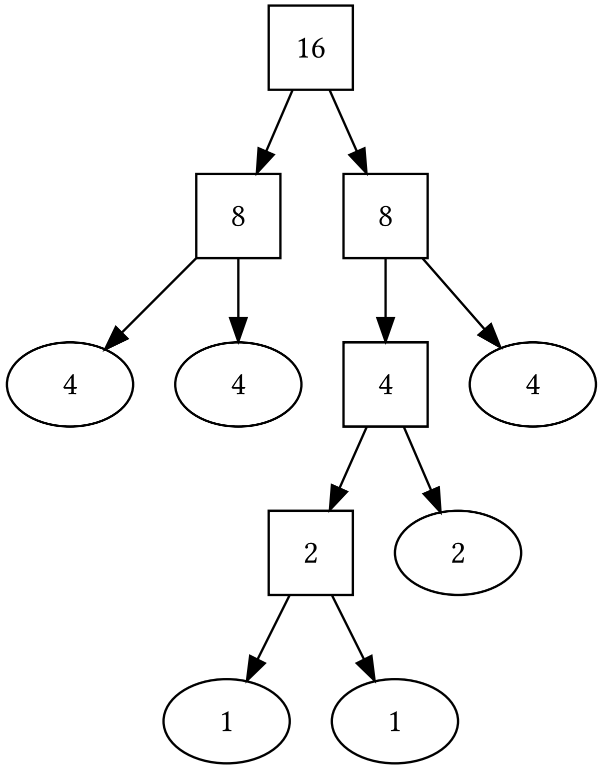
So far, what we shown can be done in *OCaml* : but we can go one step further!

```
Inductive bintree : nat -> Type :=
| leaf (x:nat) : bintree x
| node (w: nat) (l r: bintree w) : bintree
(w+w).
```

The typechecker ensures only "balanced" tree are constructed.

```
Check (node 8 (node 4 (leaf 4) (leaf 4))
(node 4 (node 2 (node 1 (leaf 1) (leaf 1))
(leaf 2)) (leaf 4))) : bintree (8 + 8) ⇒
```

This can be still done in `C++` because templates can accept integers. However, in Coq we do not have this restriction…



Term-dependent inductive type for AST

… Inductive type can depend on anything, including inductive-types!

```
Inductive Ty : Type :=
| bbool : Ty
| ffun (a b : Ty) : Ty.

(* Typing context *)
Definition TCon := list Ty.
```

```
Inductive Term : TCon -> Ty -> Type :=
| cst (tc: TCon) (b: bool) : Term tc bbool
| var (tc: TCon) (a: Ty) (p: Pos tc a)
  : Term tc a
| lam (tc: TCon) (a b: Ty)
    (tb: Term (cons a tc) b)
  : Term tc (ffun a b)
| app (tc: TCon) (a b: Ty)
  (tab: Term tc (ffun a b))
  (ta: Term tc a) : Term tc b.
```

```
Fixpoint get (tc: TCon) (a: Ty)
(p: Pos tc a) (c: Con tc) : Term nil a
```

```
Inductive Pos : TCon -> Ty -> Type :=
| here (tc: TCon) (a:Ty) : Pos (cons a tc) a
| there
  (tc: TCon) (a b: Ty) (ha: Pos tc a)
  :
  Pos (cons b tc) a.
```

```
Inductive Con : TCon -> Type :=
| cnil : Con nil
| ccons
    (a: Ty)
    (t: Term nil a)
    (tc: TCon)
    (c: Con tc)
    : Con (cons a tc).
```

```
Fixpoint eval (tc: TCon) (a: Ty) (t: Term tc
a) (c: Con tc) : Term nil a
```