Dependently typed functions

Yves Bertot

Functions returning dependent types

- Dependent datatypes
- Partial domains
- Need for dependently typed pattern-matching
  - Strong connection with induction principles
- Difficult programming: rely on tactics

Dependently typed functions

- families of types, indexed by another type A
- Described as functions of type B `:` A `-> Type`
  - also type A `-> Prop`
- Functions can return different types for different arguments
- Notation as in logic : f `: forall x :` A`,` B `x`
- Typing implies substitution
  - if e has type A, $f\ e$ is well-formed
  - the type is $f\ e$ `:` $B\ e$

Example of useful dependent types

- arrays of size n, binary words of fixed length
- Logical formulas!
  - Universally quantified theorems are functions
  - application is instanciation
  - Propositions are types, proofs are elements
- Partial functions
  - `forall x : nat, x <> 0 -> nat`
- Many more in the next lesson

Constructing dependently typed functions

Just by applying other dependently functions

```
Parameters (A : Type) (B C : A -> Type)
  (f : forall x : A, B x)
  (g : forall x : A, B x -> C x).

Definition h : forall x : A, C x := g x (f x).
```

But also through *dependent pattern matching*

Dependently typed pattern matching

- Different computations for different patterns
- Different types for different patterns
- A syntax extension to `match ... with ... end`

dependent pattern-matching explanation

```
match e as x return B x with
| p1 => e1
| p2 => e2
end
```

- The whole expression has type $B\ e$
  - replace x by e
- Each expression e_i must have type $B\ p_i$
  - replace x by p_i

Example on predecessor

```
Check False_rect.
False_rect : forall P : Type, False -> P

Print not.
not = fun A : Prop => A -> False : Prop -> Prop

Check refl_equal.
refl_equal : forall (A : Type) (x : A), x = x
```

- `False_rect` expresses that any specification can be fulfilled in an inconsistent context
- `refl_equal` is just a plain theorem, it can be used as a function

Example on predecessor (continued)

```
Definition pred_safe (x:nat) : x <> 0 -> nat :=
  match x as x return x <> 0 -> nat with
```

Example on predecessor (continued)

```
Definition pred_safe (x:nat) : x <> 0 -> nat :=
  match x as x return x <> 0 -> nat with
    0 => fun h : 0 <> 0 =>
```

Example on predecessor (continued)

```
Definition pred_safe (x:nat) : x <> 0 -> nat :=
  match x as x return x <> 0 -> nat with
    0 => fun h : 0 <> 0 =>
              False_rect nat (h (refl_equal 0))
```

Example on predecessor (continued)

```
Definition pred_safe (x:nat) : x <> 0 -> nat :=
  match x as x return x <> 0 -> nat with
    0 => fun h : 0 <> 0 =>
            False_rect nat (h (refl_equal 0))
  | S p => fun h : S p <> 0 =>  p
  end.
```

- the text in black can be forgotten: the matched expression is a variable
- `False_rect` is used to mark "unreachable code"

Dependent recursion

- In recursive definitions, dependent pattern-matching is allowed
- Calls to recursive calls are not in the same type
- This gives induction principles

Example dependent recursion

```
Fixpoint f (x : nat) : B x :=
  match x return B x with
    O => V
  | S p => E p (f p)
  end.
```

- `V` must have type `(B O)`
- `E` must have type `forall p : nat, B p -> B (S p)`
- `f` has type `forall x, B x`

Example dependent recursion (continued)

```
Fixpoint f (B : nat -> Type) (V : B 0)
  (E : forall x, B x -> B (S x)) (x : nat) : B x :=
  match x return B x with
    0 => V
  | S p => E p (f p)
  end.

Check f.
f : forall B : nat -> Type, B 0 ->
      (forall n:nat, B n -> B (S n)) ->
  forall x:nat, B x
```

- The function `f` is an induction principle

Dependent inductive types

- Families of types can be given inductively
- constructors can have dependent types
- arguments to constructors can be proofs

Bounded numbers and arrays

```
Inductive bnat (n : nat) : Type :=
  cb : forall m, m < n -> bnat n.

Inductive array (n : nat) : Type :=
  ca : forall l : list Z, length l = n -> array n.
```

- More precise than natural numbers
- Can be used to access arrays
- type-checking verifies that array bounds are respected

Using tactics

- Dependent types are like logical formulas
- Tactics can construct programs like proofs
  - `intros x` corresponds to `fun x => ...`
  - `case x` corresponds to `match x with ...end`
  - `apply f` corresponds to `f ...`

Bounded access in array

```
Definition access :
  forall (m : nat) (l : list Z), m < length l -> Z.
induction m as [ | m IHm].
2 subgoals

  ============================
   forall l : list Z, 0 < length l -> Z

subgoal 2 is:
 forall l : list Z, S m < length l -> Z
intros [ | z tl].
SearchPattern (~_ < 0).
lt_n_O: forall n : nat, ~ n < 0
```

Bounded access in array (continued)

```
intros h; case (lt_n_O _ h).
  ============================
   O < length (z :: tl) -> Z
intros _; exact z.
intros [ | z tl] h.
  h : S m < length nil
  ============================
   Z
case (lt_n_O _ h).
  IHm : forall l : list Z, m < length l -> Z
  ...
  h : S m < length (z :: tl)
  ============================
   Z
```

Bounded access in array (continued)

```
apply (IHm tl).
simpl in h.
omega.
Defined.
```

- Each step is quite easy
- Fear to loose track

Adding dependency to simply typed functions

- `if-then-else` statements, pattern-matching on boolean values
- Information gained in each branch, but not apparent in the context
- Information can be added using artificial equality arguments
- replace

  `if` e `then` e_1 `else` e_2 with

  `match` e `as b return` e `= b -> T with`

  `  true => fun h :` e `= true =>` e_1

  `| false => fun h :` e `= false =>` e_2

  `end (refl_equal` e`)`
- Done by tactic `case_eq`

Example on case_eq

```
Definition dyn_safe_access :
   forall m:nat, nat -> array m -> Z.
intros m n [l len]; case_eq (leb m n).
  =======================
   leb m n = true -> Z
intros _; exact 0%Z.
  =======================
  leb m n = false -> Z
intros h; apply (access n m).
  len : length l = m
  h : leb m n = false
  =======================
   n < length l
rewrite len; apply leb_complete_conv; exact h.
```

Bounded access in array (alternative)

```
Program Fixpoint access' (n : nat) (l:list Z)
                        : n < length l -> Z :=
  match n with
    O =>
    match l with nil => _ | z::tl => fun _ => z end
  | S p =>
    match l with
      nil => _
    | z::tl => fun _ => access' p tl _
    end
  end.
```

- Algorithmic content is explicit
- Pattern matching constructs are uncluttered

Bounded access in arrays (alternative, cont.)

```
Next Obligation.
  H : 0 < 0
  ============================
   Z
case (lt_n_O 0); assumption.
Qed.
Next Obligation.
case (lt_n_O (S p)); assumption.
Qed.
Next Obligation.
omega.
Qed.
```

Bounded access: fully specified

```
Definition safe_access : forall m, bnat m -> array m -> Z.
intros m [n h] [l len].
apply (access n l).
  rewrite len; exact h.
Defined.
```

- Array update could be described in the same way
- Loops where i goes from 1 to m can be defined
  - i with type `bnat m`