Well-founded induction

Yves Bertot

April 15, 2008

Limits of structural recursion

- Structural recursion on one argument is restrictive,
- Functions that are total, but not structural recursive are frequent,
  - merge (for merge_sort), quicksort
  - gcd,
- Structural recursion imposes the choice of data-structure,
  - factorial on Z, Q

Example of merge_sort

- merge (a::l1) (b::l2) =
  if le_lt_dec a b then a::merge l1 (b::l2)
  else b::merge (a::l1) l2
  merge nil l2 = l2
  merge l1 nil = l1
- When a $\leq$ b, only the first argument decreases
- When b $<$ a, only the second argument decreases
- The sum of length of the lists decreases by one at each recursive call!

Example of gcd

- gcd a b $=$ if a $=$ b then a else if a $<$ b then gcd a (b-a) else gcd (a-b) b
- This algorithm is guaranteed to terminate only if a and b are positive
- The sum of the two arguments decreases at each recursive call,
- The decrease is non-zero, but we can't know by how much.

bounded recursion

- Add an artificial argument to count what decreases,
- Return a dummy value when the artificial argument reaches 0,
- Ensure the artificial argument has a good initial value.

bounded recursion for merge_sort

- Fixpoint bmerge (l1 l2 : list nat)(n:nat){struct n} : list nat :=

```
match n with
  0 => nil
| S p =>
  match l1, l2 with
    (a::l1), (b::l2) =>
    if le_dec a b then a::bmerge l1 (b::l2) p
    else b::bmerge (a::l1) l2
  | nil, l2 => l2
  | l1, nil => l1
  end
end.
Definition merge (l1 l2 : list nat) :=
bmerge l1 l2 (length l1+length l2).
```

Reasoning on bounded merge

- Impose conditions on the bound to avoid the degenerate case,
- Example for merge

  Lemma merge_sorted :

  $\forall$n l1 l2, length l1 + length l2 = n $\rightarrow$

  sorted l1 $\rightarrow$ sorted l2 $\rightarrow$ sorted (merge l1 l2 n).
- Perform proofs by induction on the bound,

Drawbacks of this approach

- The artificial argument is a nuisance,
- The code developed in Coq is also meant to be transformed into software
  - Argument kept in derived code,
  - Computation of the intial value may take time.

Well-founded induction

- Support direct encoding of terminating sequences,
- Allow functions where recursive calls follow terminating sequences.

Terminating relations

- Given a relation R, an element x is accessible, if there is no infinite sequence x_n such that:
  - $x_0 = x$
  - $\forall i,\ R\ x_{i+1}\ x_i$
- Actually x is accessible if and only if all its R-predecessors are,
  Inductive Acc (A:Type)(R:A→A→Prop) : A→Prop :=
  Acc_intro : ∀x, (∀y, R y x → Acc A R y) → Acc A R x.
- A relation is well-founded when all elements are accessible.

Well-founded relations

- A Coq library makes it possible to construct well-founded relations,
- The order lt is well-founded,
- The order Zwf $\equiv$ fun a x y:Z, a $\leq$ y /\ x $<$ y is well-founded,
- Composition with a function preserves well-foundedness,
- Inclusion preserves well-foundedness,
- Lexical ordering preserves well-foundedness,
- Sometimes, one needs to prove well-foundedness by going back to accessibility.

Well-founded recursion

- A definition $f\ x = E$ has well-founded recursion for R when f only appears in E applied to expressions e such that $R\ e\ x$.
- Expressed with types, for $f\colon A \to B$
  $F : \forall x{:}A, (\forall y{:}A, R\ y\ x \to B) \to B$
  $f\ x = F\ x\ f$
- More generally, with dependent types, for $f : \forall x{:}A, B\ x$
  $F : \forall x{:}A, (\forall y{:}A, R\ y\ x \to B\ y) \to B\ x$

Well-founded recursion: the Fix construct

- Fix maps any F to the corresponding f
  Fix : $\forall$(A : Type) (R : A $\rightarrow$ A $\rightarrow$ Prop), well_founded R $\rightarrow$
  $\forall$P : A $\rightarrow$ Type,
  ($\forall$x : A, ($\forall$y : A, R y x $\rightarrow$ P y) $\rightarrow$ P x) $\rightarrow$
  $\forall$x : A, P x
- Good to choose a dependent type for P

Example for merge

- Input type : list nat*list nat,
- measure : m $\equiv$ fun a => length (fst a) + length (snd a)
- Relation : R a b $\equiv$ m a < m b
- output specification:
  Q a l $\equiv$
  permutation l (fst a++snd a) /\
  sorted (fst a) $\rightarrow$ sorted (snd a) $\rightarrow$ sorted l}
- output type : P a $\equiv$ {l:list nat | Q a l}

Well-founded recursion for merge (continued)

```
mergeF x f ≡
match x return P x with
  (a::l1,b::l2) =>
     match le_lt_dec a b with
       left ab =>
       let (l, lp) := f (l1, b::l2) (dec1 a l1 b l2) in
       exist (Q (a::l1,b::l2)) (a::l) (rp1 a l1 b l2 l ab lp)
     | right ba =>
       let (l, lp) := f (a::l1, l2) (dec2 a l1 b l2) in
       exist (Q (a::l1,b::l2)) (b::l) (rp2 a l1 b l2 ba lp)
     end.
| (nil, l2) => exist (Q (nil, l2)) l2 (rp3 l2)
| (l1, nil) => exist (Q (l1, nil)) l1 (rp4 l1)
end.
```

Well-founded recursion for merge (continued)

- Lemma dec1 : $\forall$a l1 b l2, m (l1, b::l2) $<$ m (a::l1, b::l2).
- Lemma rp1 : $\forall$a l1 b l2 l, a $\leq$ b $\rightarrow$
  Q (l1, b::l2) l $\rightarrow$ Q (a::l1, b::l2) (a::l)

Well-founded recursion for merge (continued)

- Definition merge = Fix (Wf_nat.well_founded_ltof _ m) mergeF
- Almost no need to do proofs about this code: information in the type.
- If behavior must be analyzed, use theorem Fix_eq.

The Function command

```
Function merge (p:list nat * list nat) {measure m} : list nat :=
match p with (a::l1, b::l2) =>
 if le_lt_dec a b then a::merge (l1,b::l2) else b::merge(a::l1, l2)
| (nil, l2) => l2
| (l1, nil) => l1
end.
```

- The command produces goals, which correspond to dec1 and dec2 of the previous slide,
- The correctness with respect to sorting and permutation is not proved yet,
- There is a specific induction principle for the function (too verbose to show here, but very useful for the proofs) (make a demo).