DD2552 Seminar 6: Proving properties of functions

Karl Palmskog

KTH

Thursday September 14, 2023

Course material

- "Some notes on structural induction" (https://www.cs.cmu.edu/~me/courses/15-150-Spring2020/lectures/04/structural.pdf)
- "Proving properties of programs by structural induction" by Burstall, 1968

Premises for this seminar

- we assume data is inductive (not coinductive)
- we assume recursive functions are **least** fixpoints (not greatest)
- we get Harper's System FPC with eager dynamics
- this is close to Standard ML, OCaml or CakeML

Reminder on ensuring function termination

- come up with measure on function input (arguments)
- define relation over two measures
- prove relation has no infinitely descending chains (is wellfounded)
- show that measure decreases in all recursive calls

Proving other properties than termination

- functions are defined recursively on structure of data
- use induction on structure of data to prove properties
- we get one case for each "data constructor" and hypotheses for subterms

Proving equalities

- many interesting properties are equalities
- thanks to confluence, we can substitute terms for reduced terms
- we can also reason in "point-free" style
  - no function arguments in the way
  - would need to assume **extensional equality**
- hypothesis: substituting equals-for-equals is the core property that makes reasoning about pure/stateless functions feasible and practical

Appending lists of integers in CakeML

```
datatype list = Nil | Cons int list

fun app l1 l2 =
 case l1 of
   Nil => l2
 | Cons a l11 => Cons a (app l11 l2)
```

Property:

```
app (app l1 l2) l3 = app l1 (app l2 l3)
```

Reversing lists of integers in CakeML

```
fun reverse l1 =
 case l1 of
   Nil => Nil
 | Cons a l11 => app (reverse l11) (Cons a Nil)

fun rev_aux l1 l2 =
 case l1 of
   Nil => l2
 | Cons a l11 => rev_aux l11 (Cons a l2)

fun rev l1 = rev_aux l1 Nil
```

Simpler functions as specifications

- we can use `reverse` as specification of what list reversal means
- we can view `rev` as a proposed optimization
- prove for all `l` that `reverse l = rev l`

Functional depth-first search

```
Fixpoint dfs n v x :=
  if x \in v then v else
  if n is n'.+1 then foldl (dfs n') (x :: v) (g x) else v.
```

Functional merge sort

```
Fixpoint merge s1 :=
if s1 is x1 :: s1' then
  let fix merge_s1 s2 :=
    if s2 is x2 :: s2' then
      if leT x1 x2 then
       x1 :: merge s1' s2
      else x2 :: merge_s1 s2'
    else s1 in merge_s1
else id.

Fixpoint merge_sort_push s1 ss :=
match ss with
| [::] :: ss'
| [::] as ss' => s1 :: ss'
| s2 :: ss' => [::] :: merge_sort_push (merge s2 s1) ss'
end.
```

Merge sort, continued

```
Fixpoint merge_sort_pop s1 ss :=
if ss is s2 :: ss' then
 merge_sort_pop (merge s2 s1) ss' else s1.

Fixpoint merge_sort_rec ss s :=
if s is [:: x1, x2 & s'] then
  let s1 :=
   if leT x1 x2 then
    [:: x1; x2]
   else [:: x2; x1]
  in
  merge_sort_rec (merge_sort_push s1 ss) s'
else merge_sort_pop s ss.

Definition sort := merge_sort_rec [::].
```