5th Asian-Pacific Summer School on Formal Methods

August 5-10, 2013, Tsinghua University, Beijing, China

Program Verification in Coq

Guillaume Melquiond

August 9th, 2013

http://sts.thss.tsinghua.edu.cn/Coqschool2013

Cheat Sheet (1/4)

Things that are always good to do:

- When an assumption states `x <> x`, change the goal to `False` using `exfalso`, then conclude.
- When two assumptions are in contradiction, change the goal to `False` using `exfalso`, then conclude.
- When an assumption states `C1 ... = C2 ...` with `C1` and `C2` two different constructors, `discriminate` it.
- When the goal is an equality `x = x`, use `reflexivity`.
- When the goal is `True`, `apply I`.

Cheat Sheet (2/4)

Things that are (almost) always good to do:

- When the goal is a forall, an implication, a negation, introduce its left-hand side with `intros`.
- When an assumption is a conjunction or an inductive object with a single constructor (e.g. a pair), `destruct` it.
- When the goal is a disjunction, select the provable side using `left` and `right` as soon as you know it.
- Perform computations with `simpl`, or with `change` if `simpl` goes too far.

Cheat Sheet (3/4)

Things that are good to do, but as late as possible:

- When the goal is a conjunction, `split` it.
- When an assumption is a disjunction or an inductive object with several constructors, `destruct` it.

Cheat Sheet (4/4)

Things to do in the remaining cases:

- When the goal contains an application `f x` with `f` a fixpoint definition, perform an `induction` on `x`.
- Before doing the `induction`, `revert` all the arguments that are not constant in the recursive call of `f`.
- When the goal contains a `match` on a value, `destruct` it.
- Do `apply` lemmas or `rewrite` with equalities.

Some Simple Functions on Lists

```
Definition head {T : Type} (l : list T) : option T :=
  match l with
  | nil => None
  | cons h _ => Some h
  end.

Definition tail {T : Type} (l : list T) : list T :=
  match l with
  | nil => nil
  | cons h q => q
  end.
```

Accessing the *n*-th Element of a List

```
Fixpoint get {T : Type} (l : list T) (n : nat)
         {struct l} : option T :=
  match l with
  | nil => None
  | cons h q =>
    if n == 0 then Some h else get q (n - 1)
  end.
```

Modifying the *n*-th Element of a List

```
Fixpoint set {T : Type} (l : list T) (n : nat) (v : T)
         {struct l} : list T :=
  match l with
  | nil => l
  | cons h q =>
    if n == 0 then cons v q
    else cons h (set q (n - 1) v)
  end.
```

Note: the original list is not modified; a new list is returned.

Time Complexity for Standard Lists

Time complexity: how many lists have to be constructed / destructed in order to perform a given operation.

- `cons: T -> list T -> list T` $O(1)$
- `head: list T -> option T` $O(1)$
- `tail: list T -> list T` $O(1)$
- `get : list T -> nat -> option T` $O(n)$
- `set : list T -> nat -> T -> list T` $O(n)$

Note: `get` and `set` are slow!

Random Access Lists (Chris Okasaki)

Time complexity: how many lists have to be constructed / destructed in order to perform a given operation.

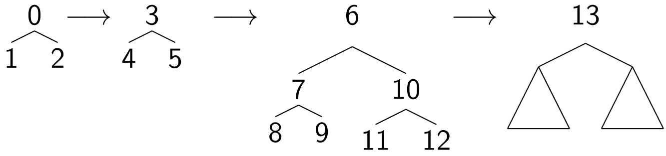
- `racons:  T -> ralist -> ralist` $O(1)$
- `rahead:  ralist -> option T` $O(1)$
- `ratail:  ralist -> ralist` $O(1)$
- `raget :  ralist -> nat -> option T` $O(\log n)$
- `raset :  ralist -> nat -> T -> ralist` $O(\log n)$

Note: `get` and `set` went from $O(n)$ to $O(\log n)$.

Random Access Lists (Chris Okasaki)

Internal representation:

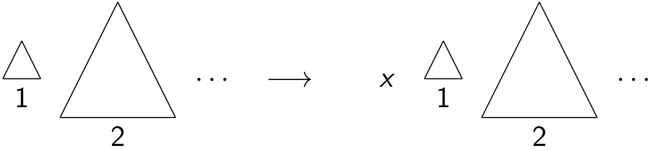
- List of balanced trees with nodes labeled by elements of `T`.
- Trees of the list have strictly increasing heights.
  Exception: the first two trees may have the same height.
- The older the elements, the further in the list of trees they are.
  Tree elements are stored with a depth-first pre-order traversal.



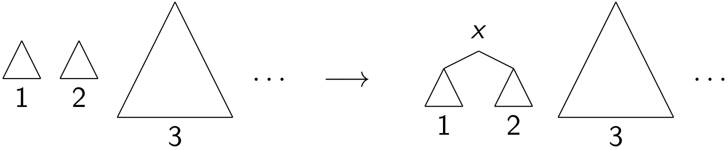
Note: the reduced complexity comes from the fact that $2n$ operations suffices to access the 2^n first elements.

Adding an Element to a RA List

- If the first two trees have different heights,



- If the first two trees have the same height,



Coq Types for Representing RA Lists

```
Variable T : Type.

Inductive tree :=
  | Leaf : T -> tree
  | Node : T -> tree -> tree -> tree.

Inductive ralist :=
  | raNil : ralist
  | raCons : tree -> nat -> ralist -> ralist.
```

Note: `raCons` stores a tree, its height, and the remaining of the list.

Definition of Head

```
Definition rahead (l : ralist) : option T :=
  match l with
  | raNil => None
  | raCons t _ _ =>
    match t with
    | Leaf x => Some x
    | Node x _ _ => Some x
    end
  end.
```

Correctness of Head

In order to verify that `rahead` is correct,
one has to prove that it has the same behavior as `head`.

```
Definition abs : ralist -> list T := ...

Lemma rahead_correct :
  forall l : ralist,
  rahead l = head (abs l).
```

Abstracting from RA Lists to Standard Lists

```
Fixpoint abs_tree (t : tree) {struct t} : list T :=
  match t with
  | Leaf x => cons x nil
  | Node x t1 t2 =>
    cons x (app (abs_tree t1) (abs_tree t2))
  end.

Fixpoint abs (l : ralist) {struct l} : list T :=
  match l with
  | raNil => nil
  | raCons t _ q => app (abs_tree t) (abs q)
  end.
```

Definition and Correctness of Cons

```
Definition racons (x : T) (l : ralist) : ralist :=
  match l with
  | raNil => raCons (Leaf x) 0 l
  | raCons t s raNil => raCons (Leaf x) 0 l
  | raCons t1 h1 (raCons t2 h2 q) =>
    if h1 == h2 then raCons (Node x t1 t2) (1 + h1) q
    else raCons (Leaf x) 0 l
  end.

Lemma racons_correct :
  forall (x : T) (l : ralist),
  abs (racons x l) = cons x (abs l).
```

Definition and Correctness of Tail

```
Definition ratail (l : ralist) : ralist :=
  match l with
  | raNil => raNil
  | raCons t h q =>
    match t with
    | Leaf _ => q
    | Node _ t1 t2 =>
      raCons t1 (h - 1) (raCons t2 (h - 1) q)
    end
  end.

Lemma ratail_correct :
  forall l : ralist,
  abs (ratail l) = tail (abs l).
```

Summary

What was done:

- Defining `tree` and `list`.
- Defining `rahead`, `racons`, and `ratail`.
- Proving that they behave like `head`, `cons`, and `tail`, according to the `abs` mapping.

What has not be done yet:

- Proving that `racons` and `ratail` produce trees that are both balanced and of (strictly) increasing height.
- Defining `raget` and `raset`.
- Proving that they are correct.

Data Invariant

```
Fixpoint height (t : tree) {struct t} : nat :=
  match t with
  | Leaf _ => 0
  | Node _ t1 _ => 1 + height t1
  end.

Fixpoint balanced (t : tree) {struct t} : Prop :=
  match t with
  | Leaf _ => True
  | Node _ t1 t2 =>
    height t1 = height t2 /\
    balanced t1 /\ balanced t2
  end.
```

Note: `height` assumes that the tree is balanced.

Data Invariant

```
Fixpoint structured_aux (l : ralist) (h : nat)
         {struct l} : Prop :=
  match l with
  | raNil => True
  | raCons t h' q =>
    balanced t /\ height t = h' /\ h <= h' /\
    structured_aux q (1 + h')
  end.

Definition structured (l : ralist) : Prop :=
  match l with
  | raNil => True
  | raCons t h q =>
    balanced t /\ height t = h /\
    structured_aux q h
  end.
```

Note: these are functional predicates, rather than inductive ones.

Preservation of Invariant

```
Lemma structured_racons :
  forall (l : ralist) (x : T),
  structured l ->
  structured (racons x l).

Lemma structured_ratail :
  forall (l : ralist),
  structured l ->
  structured (ratail l).
```

Definition of Get

```
Fixpoint tree_get (t : tree) (h : nat) (n : nat)
         {struct t} : option T :=
  match t with
  | Leaf x => if n == 0 then Some x else None
  | Node x t1 t2 =>
    if n == 0 then Some x
    else
      let s := height2size (h - 1) in
      if n <= s then tree_get t1 (h - 1) (n - 1)
      else tree_get t2 (h - 1) (n - 1 - s)
  end.

Fixpoint raget (l : ralist) (n : nat)
         {struct l} : option T :=
  match l with
  | raNil => None
  | raCons t h q =>
    let s := height2size h in
    if n < s then tree_get t h n
    else raget q (n - s)
  end.
```

Code Extraction

Principles:

1. Write a library in Coq.
2. Prove its correctness using Coq.
3. Extract it to a functional language, e.g. OCaml or Haskell.
4. Profit!

Code Extraction

- Map Coq types to types from the target language:

```
Extract Inductive bool =>
  "bool" [ "true" "false" ].
Extract Inductive option =>
  "option" [ "Some" "None" ].
Extract Inductive nat => "int" [ "0" "succ" ]
  "(fun fO fS n ->
    if n=0 then fO () else fS (n-1))".
```

Note: the mapping of `nat` is unsafe.

- Map Coq functions:

```
Extract Inlined Constant leb => "(<=)".
Extract Inlined Constant eqb => "(==)".
Extract Inlined Constant plus => "(+)".
Extract Inlined Constant minus => "(-)".
```

Note: the mapping of `minus` is terribly wrong.