Verification of Imperative Programs Through Characteristic Formulae

Arthur Charguéraud

INRIA

Microsoft Research

Cambridge, 2010/11/18

The big picture

Caml

Source code not annotated

Automatic generation →

Coq

Characteristic formulae

Interactive proofs ↓

Specification & verification

Characteristic formulae

Total correctness Hoare triple: under the pre-condition H, the term t terminates and produces a value v such that (Q v) describes the post-condition.

$$\{H\}\ t\ \{Q\}$$

Characteristic formula, written $[\![t]\!]$, such that:

$$[\![t]\!]\ H\ Q \quad\Longleftrightarrow\quad \{H\}\ t\ \{Q\}$$

higher-order logic predicate, pretty-printed like the term *t*

program syntax

Characteristic formula : Hprop → (T → Hprop) → Prop, where Hprop = Heap → Prop.

CF for let-expressions

Hoare logic:

$$\frac{\{H\}\; t_1\; \{Q'\} \qquad \forall x.\ \{Q'\,x\}\; t_2\; \{Q\}}{\{H\}\; (\mathsf{let}\; x = t_1\; \mathsf{in}\; t_2)\; \{Q\}}$$

Characteristic formula:

$$[\![\mathsf{let}\; x = t_1\; \mathsf{in}\; t_2]\!] \quad \equiv$$
$$\lambda H.\, \lambda Q.\ \ \exists Q'.\ [\![t_1]\!]\, H\, Q' \ \wedge\ \forall x.\ [\![t_2]\!]\, (Q'\,x)\, Q$$

Introduction of notation:

$$(\mathbf{let}\; x = \mathcal{F}_1\; \mathbf{in}\; \mathcal{F}_2) \quad \equiv$$
$$\lambda H.\, \lambda Q.\ \ \exists Q'.\ \mathcal{F}_1\, H\, Q' \ \wedge\ \forall x.\ \mathcal{F}_2\, (Q'\,x)\, Q$$

Characteristic formula generator:

$$[\![\mathsf{let}\; x = t_1\; \mathsf{in}\; t_2]\!] \quad \equiv \quad (\mathbf{let}\; x = [\![t_1]\!]\; \mathbf{in}\; [\![t_2]\!])$$

CF generation

A similar trick applies for other constructions:

$$
\begin{array}{lcl}
[\![v]\!] & \equiv & \mathbf{return}\ v \\
[\![f\ v]\!] & \equiv & \mathbf{app}\ f\ v \\
[\![\mathsf{crash}]\!] & \equiv & \mathbf{crash} \\
[\![\mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2]\!] & \equiv & \mathbf{if}\ v\ \mathbf{then}\ [\![t_1]\!]\ \mathbf{else}\ [\![t_2]\!] \\
[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2]\!] & \equiv & \mathbf{let}\ x\ =\ [\![t_1]\!]\ \mathbf{in}\ [\![t_2]\!] \\
[\![\mathsf{let\ rec}\ f\ x = t_1\ \mathsf{in}\ t_2]\!] & \equiv & \mathbf{let\ rec}\ f\ x = [\![t_1]\!]\ \mathbf{in}\ [\![t_2]\!] \\
[\![\mathsf{while}\ t_1\ \mathsf{do}\ t_2]\!] & \equiv & \mathbf{while}\ [\![t_1]\!]\ \mathbf{do}\ [\![t_2]\!] \\
[\![\mathsf{for}\ i = a\ \mathsf{to}\ b\ \mathsf{do}\ t]\!] & \equiv & \mathbf{for}\ i = a\ \mathbf{to}\ b\ \mathbf{do}\ [\![t]\!]
\end{array}
$$

CF: easy to generate, compositional, easy to read

$$\{H\}\ t\ \{Q\} \quad \Longleftrightarrow \quad [\![t]\!]\ H\ Q \quad \Longleftrightarrow \quad \mathbf{t}\ H\ Q$$

Integration of the frame rule

Hoare logic:

$$\frac{\{H_1\}\ t\ \{Q_1\}}{\{H_1 * H_2\}\ t\ \{Q_1 \star H_2\}}$$

where $Q_1 \star H_2$ is defined as "$\lambda v.\,(Q_1\, v) * H_2$"

Updated definition:

$$\begin{aligned}(\textbf{let } x = \mathcal{F}_1 \textbf{ in } \mathcal{F}_2) \quad &\equiv \\ &\text{frame}\,(\lambda H.\,\lambda Q.\ \ \exists Q'.\ \mathcal{F}_1\, H\, Q' \ \wedge\ \forall x.\ \mathcal{F}_2\,(Q'\,x)\,Q)\end{aligned}$$

Predicate presentation:

$$\text{frame}\,\mathcal{F} \quad \equiv \quad \lambda H\, Q.\ \exists H_1\, H_2\, Q_1.\ \begin{cases} H = H_1 * H_2 \\ \mathcal{F}\, H_1\, Q_1 \\ Q = Q_1 \star H_2 \end{cases}$$

The predicate "local" generalizes "frame". It supports the rules of consequence and of garbage collection, as well as extraction of quantifiers and propositions.

Translation of types

The Caml type T is reflected as the Coq type ⟨T⟩

$$
\begin{array}{lcl}
\langle \mathsf{int} \rangle & \equiv & \mathsf{Int} \\
\langle \tau_1 \times \tau_2 \rangle & \equiv & \langle \tau_1 \rangle \times \langle \tau_2 \rangle \\
\langle \tau_1 + \tau_2 \rangle & \equiv & \langle \tau_1 \rangle + \langle \tau_2 \rangle \\
\langle \tau_1 \rightarrow \tau_2 \rangle & \equiv & \mathsf{Func} \\
\langle \mathsf{ref}\, \tau \rangle & \equiv & \mathsf{Loc}
\end{array}
$$

– A value of type **Func** corresponds to the source code of a well-typed Caml function

– A **Heap** is a map from type **Loc** to dependent pairs made of a type and a value of that type

Observe: no negative-occurence of recursive types

CF for function applications

AppReturns f v H Q states that the application of f to v admits pre-condition H and post-condition Q.

Example: (App is the same as AppReturns for arity 1)

```
(App incr r;) (r ~~> n) (fun _ => r ~~> n+1)
```

Type of AppReturns:

$$\forall A\, B.\ \mathsf{Func} \to A \to \mathsf{Hprop} \to (B \to \mathsf{Hprop}) \to \mathsf{Prop}$$

Characteristic formulae for applications:

$$[\![f\ v]\!]\ H\ Q \quad \Leftrightarrow \quad \mathsf{AppReturns}\ f\ v\ H\ Q$$

$$[\![f\ v]\!] \quad \equiv \quad \mathsf{AppReturns}\ f\ v$$

CF for function definitions

Instances of the predicate AppReturns:

– have to be provided for reasoning on applications

– are given by the formula of a function definition

Instances generated for $\mathsf{let\ rec}\ f\ x = t$

$$\forall x\, H'\, Q'.\ [\![t]\!]\, H'\, Q' \ \Rightarrow\ \mathsf{AppReturns}\, f\, x\, H'\, Q'$$

Characteristic formulae for functions:

$$[\![\mathsf{let}\ f\ x = t\ \mathsf{in}\ t']\!] \quad \equiv$$
$$\lambda HQ.\forall f.\ (\forall x\, H'\, Q'.\ [\![t]\!]\, H'\, Q' \ \Rightarrow\ \mathsf{AppReturns}\, f\, x\, H'\, Q') \ \Rightarrow\ [\![t']\!]\, H\, Q$$

Recursive functions are proved correct by induction

Characteristic formula generation

$H_1 \rhd H_2$ is **H1 ==> H2**

$$[\![v]\!] \equiv \text{local}\,(\lambda H Q.\ H \rhd Q\, v)$$

$$[\![f\ v]\!] \equiv \text{local}\,(\lambda H Q.\ \text{AppReturns}\ f\ v\ H\ Q)$$

$$[\![\text{crash}]\!] \equiv \text{local}\,(\lambda H Q.\ \text{False})$$

$$[\![\text{if}\ v\ \text{then}\ t_1\ \text{else}\ t_2]\!] \equiv \text{local}\,(\lambda H Q.\ (v = \text{true} \Rightarrow [\![t_1]\!]\, H\, Q) \wedge (v = \text{false} \Rightarrow [\![t_2]\!]\, H\, Q))$$

$$[\![\text{let}\ x = t_1\ \text{in}\ t_2]\!] \equiv \text{local}\,(\lambda H Q.\ \exists Q'.\ [\![t_1]\!]\, H\, Q' \wedge \forall x.\ [\![t_2]\!]\, (Q'\, x)\, Q)$$

$$[\![\text{let}\ f\ x = t_1\ \text{in}\ t_2]\!] \equiv \text{local}\,(\lambda H Q.\ \forall f.\ \mathcal{H} \Rightarrow [\![t_2]\!]\, H\, Q)$$

where $\mathcal{H}$ is $(\forall x\, H'\, Q'.\ [\![t_1]\!]\, H'\, Q' \Rightarrow \text{AppReturns}\ f\ x\ H'\, Q')$

Incrementation function

Caml source code:

```
let incr r =
  r := !r + 1
```

Normalized Caml code:

```
let incr r =
  let x = get r in
  set r (x+1)
```

Generated Coq definitions:

```
Axiom incr : Func.
Axiom incr_cf : ∀ (r:loc) (H:Hprop) (Q:unit->Hprop),
  (Let x = App get r; in App set r (x+1)) H Q ->
  (App incr r;) H Q.
```

behind the scene: ∧, ⇒, ∀, ∃, ...

Incr: verification (1/2)

```
Lemma incr_spec : ∀ (r:loc) (n:int),
  (App incr r;) (r ~~> n) (fun _ => r ~~> n+1).
                  Hprop         unit → Hprop

Proof. xcf. xlet. xapp. xextract. xapp. xsimpl. Qed.
```

xcf

```
(r:loc) (n:int)
|- (Let x = App get r; in App set r (x+1);)
     (r ~~> n) (# r ~~> n+1).
```

xlet

```
(r:loc) (n:int)
|- (App get r) (r ~~> n) ?Q.

(r:loc) (n:int) (x:int)
|- (App set r (x+1);) (?Q x) (# r ~~> n+1).
```

xapp

```
?Q = (fun a => [a = n] \* r ~~> n)
```

```
Lemma get_spec : ∀ (A:Type) (r:loc) (v:A),
  (App get r;) (r ~~> v) (fun a => [a = v] \* r ~~> v)
```

Incr: verification (2/2)

```
(r:loc) (n:int) (x:int)
|- (App set r (x+1);) ([x = n] \* r ~~> n) (# r ~~> n+1).
```

```
(r:loc) (n:int) (x:int) (H: x = n)
|- (App set r (x+1)) (r ~~> n) (# r ~~> n+1).
```

xextract

```
(r:loc) (n:int) (x:int) (H: x = n)
|- (r ~~> x+1) ==> (r ~~> n+1).
```

xapp

```
(r:loc) (n:int) (x:int) (H: x = n)
|- (x+1) = (n+1)
```

xsimpl

```
Lemma set_spec : ∀ (A:Type) (r:loc) (u:A) (v:A),
  (App set r v;) (r ~~> u) (# r ~~> v).
```

Incr: summary

Specification:

```
Lemma incr_spec : ∀ (r:loc) (n:int),
  (App incr r;) (r ~~> n) (# r ~~> n+1).
```

Verification:

```
Proof. xcf. xlet. xapp. xextract. xapp. xsimpl. Qed.
```

```
Proof. xcf. xapp. xapp. xsimpl. Qed.
```

```
Proof. xgo*. Qed.
```

Specification with "Spec" notation:

```
Lemma incr_spec :
  Spec incr (r:int) |R>> ∀n, R (r ~~> n) (# r ~~> n+1).
```

Automated framing

```
(a:loc) (x:int) (b:loc) (y:int)
|- (App incr b;) (a ~~> x \* b ~~> y) ?Q
```

```
xapp: should unify  ?Q  with  # (a ~~> x \* b ~~> y+1)
```

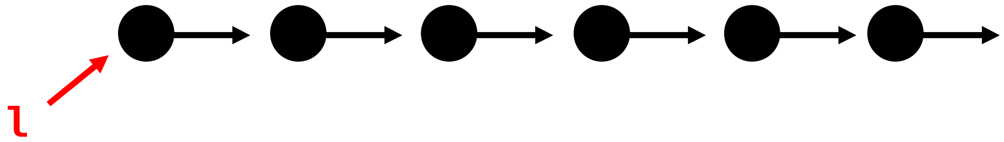
```
Lemma incr_spec :
  Spec incr (r:int) |R>> ∀n, R (r ~~> n) (# r ~~> n+1)
```

```
                              R (b ~~> ?n) (# b ~~> ?n+1)
```

```
(1) (a ~~> x) \* (b ~~> y) ==> (b ~~> ?n) \* ?H

(2) (# b ~~> ?n+1) \*+ ?H ==> ?Q
```

```
From (1) deduce  ?n = y  and  ?H = (a ~~> x)

Then (2) becomes
  (# b ~~> y+1) \*+ (a ~~> x) ==> ?Q

Thus we deduce ?Q = # (b ~~> y+1 \* a ~~> x)
```

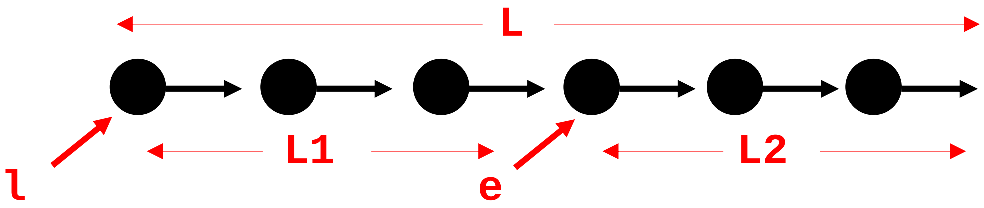
Length of mutable lists




```
> type 'a mlist = { hd : 'a; tl : 'a mlist } (*or null*)
> let mlength (l:'a mlist) =
>   let h = ref l in
>   let n = ref 0 in
>   while !h != null do
>     incr n;
>     h := !h.tl;
>   done;
>   !n
```

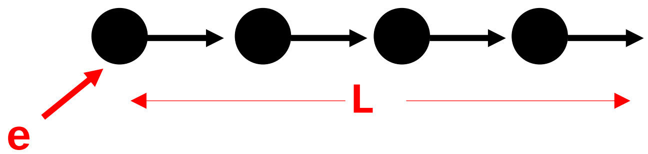
```
Lemma mlength_spec : ∀(A:Type),
  Spec mlength (l:loc) |R>> ∀(L:list A),
    keep R (l ~> MList L) (\= length L)

    R (l ~> MList L) (fun x => [x = length L] \* l ~> MList L)
```

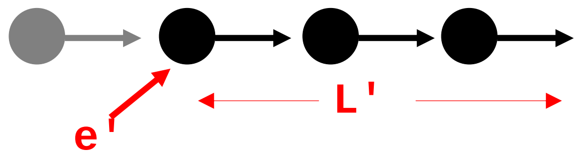
Loop invariants... or not



Loop invariant: (indexed by L2)

```
fun L2 =>
  Hexists L1 e,
  [L = L1 ++ L2] \* (n ~~> length L1) \* (h ~~> e)
  \* (l ~> MListSeg e L1) \* (e ~> MList L2))
```





Recursive presentation:

```
forall L e k,
R ((h ~~> e) \* (e ~> Mlist L) \* (n ~~> k))
(# (h ~~> null) \* (e ~> MList L) \* (n ~~> k+length L))
```

Verification of mlength

```
Lemma mlength_spec : forall a,
  Spec mlength (l:mlist a) |R>> forall A (T:A->a->Hprop) (L:list A),
    keep R (l ~> MList T L) (\= length L).
Proof.
  xcf. intros. xapp. xapp.
  xwhile (forall L l k,
    R (n ~~> k \* h ~~> l \* l ~> MList T L)
      (# n ~~> (k + length L) \* h ~~> null \* l ~> MList T L)).
   applys (>> Inv l). hsimpl.
   clear l L. intros L. induction_wf IH: (@list_sub_wf A) L; intros.
   applys (rm HR). xlet. xapps. xapps. xifs.
   (* case cons *)
   xchange (MList_not_null l) as x l' X L' EL. auto.
   xapps. xapps. xapps. xapp. subst L. xapplys~ (>> IH L' l').
   hsimpl. intros _. hchanges (MList_uncons l). rew_length. math.
   (* case nil *)
   subst. xchange MList_null_keep as M. subst.
    xrets. rew_length. math.
  xapp. hsimpl~.
Qed.
```

CF for loops, with invariants

For-loop: invariant of type "int $\rightarrow$ Hprop"

$$[\![\text{for } i = a \text{ to } b \text{ do } t_1]\!] \;\equiv$$

$$\text{local}\,(\lambda HQ.\ \exists I.\ \left\{\begin{array}{l} H \rhd I\,a \\ \forall i \in [a,b].\ [\![t_1]\!]\,(I\,i)\,(\#\,I\,(i+1)) \\ I\,(\max a\,(b+1)) \rhd Q\,tt \end{array}\right. \;)$$

While-loop: invariants of type "A $\rightarrow$ Hprop" and of type "A $\rightarrow$ bool $\rightarrow$ Hprop", for some type A.

$$[\![\text{while } t_1 \text{ do } t_2]\!] \;\equiv\; \text{local}\,(\lambda HQ.$$

$$\exists A.\ \exists I.\ \exists J.\ \exists(\prec).\ \left\{\begin{array}{l} \text{well-founded}(\prec) \\ \exists X_0.\ H \rhd I\,X_0 \\ \forall X.\ [\![t_1]\!]\,(I\,X)\,(J\,X) \\ \forall X.\ [\![t_2]\!]\,(J\,X\,\text{true})\,(\#\,\exists\!\!\!\exists Y.\ (I\,Y) * [Y \prec X]) \\ \forall X.\ J\,X\,\text{false} \rhd Q\,tt \end{array}\right.$$

CF for loops, recursive style

While-loop: (R : Hprop → (unit → Hprop) → Prop)

$$\llbracket \mathsf{while}\, t_1\, \mathsf{do}\, t_2 \rrbracket \;\equiv$$
$$\mathsf{local}\,(\lambda HQ.\;\; \forall R.\; \mathsf{is\_local}\, R \,\wedge\, \mathcal{H} \;\Rightarrow\; R\,H\,Q)$$

$$\text{with } \mathcal{H} \;\equiv\; \forall H'Q'.\; \llbracket \mathsf{if}\, t_1\, \mathsf{then}\,(t_2\,;\,|R|)\,\mathsf{else}\, tt \rrbracket\, H'\, Q' \;\Rightarrow\; R\,H'\,Q'$$

For-loop: (S : int → Hprop → (unit → Hprop) → Prop)

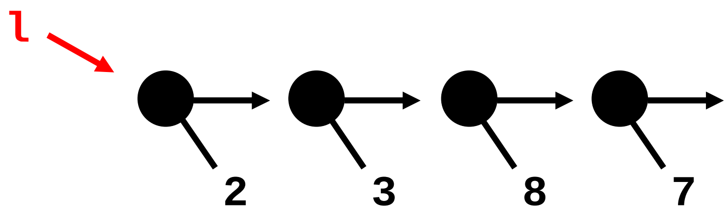
$$\llbracket \mathsf{for}\, i = a\, \mathsf{to}\, b\, \mathsf{do}\, t \rrbracket \;\equiv$$
$$\mathsf{local}\,(\lambda HQ.\;\; \forall S.\; \mathsf{is\_local}_1\, S \,\wedge\, \mathcal{H} \;\Rightarrow\; S\,a\,H\,Q)$$

$$\text{with } \mathcal{H} \;\equiv$$
$$\forall i H'Q'.\; \llbracket \mathsf{if}\, i \le b\, \mathsf{then}\,(t\,;\,|S\,(i+1)|)\,\mathsf{else}\, tt \rrbracket\, H'\, Q' \;\Rightarrow\; S\,i\,H'\,Q'$$

Representation predicates

A representation predicate T relates a Caml value with its Coq representation

```
Lemma mlength_spec : ∀(a:Type),
  Spec mlength (l:loc) |R>>
    ∀ (A:Type) (T:A->a->Hprop) (L:list A),
    keep R (l ~> MList T L) (\= length L)
```



`l ~> MList Id (2::3::8::7::nil)`

List of integers (a = A = int): l ~> MList Id_{int} L

where L : list int and L = 2::3::8::7::nil

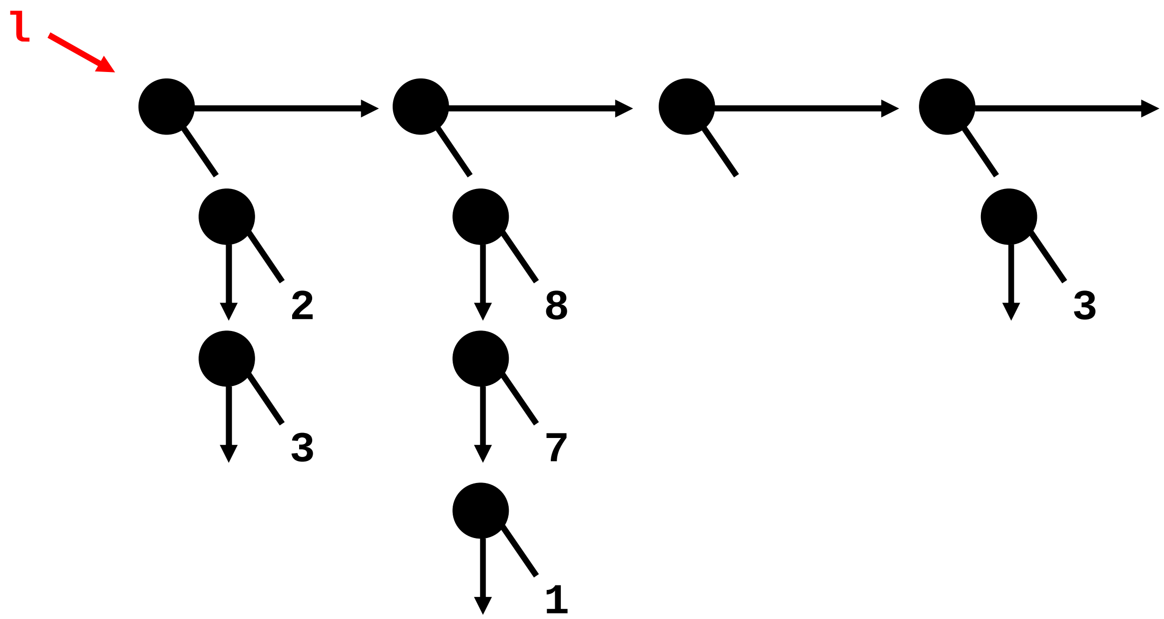
```
Definition Id (A:Type) (x:A) (X:A) : hprop := [x = X].
```

Recursive ownership

List of lists of integers: l ~> Mlist (Mlist Id_{int}) L

where L : list (list int)

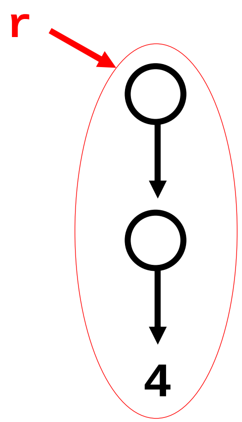
and L = (2::3::nil)::(8::7::1::nil)::nil::(3::nil)::nil



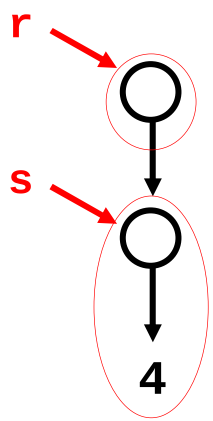
Recursive ownership is useful to describe tree-shaped mutable data structures

Reference on a reference

r ~> Ref (Ref Id_{int}) 4



r ~> Ref Id_{loc} s
* s ~> Ref Id_{int} 4



Reading and writing is restricted to "Id" fields:

```
Spec get r |R>> ∀v, keep R (r ~> Ref Id v) (\= v)
```

(r ~~> v)

Conversion lemma:

```
(r ~> Ref T X) = (Hexists x, r ~> Ref Id x \* x ~> T X)
```

Representation predicates

Tagged application:

```
x ~> U   <=>   hdata U x   <=>   U x
```

Definition of "Ref":

```
Definition Ref (a A:Type) (T:A->a->Hprop) (l:loc) :=
  Hexists x,    (heap_is_single l (ref_record x))
             \* (x ~> T X)
```

Generated material for records:

– representation predicate,

– conversion lemmas

– create, get and set specifications

Conversion lemmas for MList

```
Fixpoint MList a A (T:A->a->hprop) (L:list A) (l:loc) :=
  match L with
  | nil => [l = null]
  | X::L' => l ~> Mlist T (MList T) X L'
  end.
```

Conversion lemmas:

```
(l ~> MList T nil) = [l = null]

(l ~> MList T (X::L)) =
  (Hexists x t, l ~> Mlist Id Id x t
             \* x ~> T X
             \* t ~> MList T L)

(null ~> MList T L) = [L = nil]
```

etc...

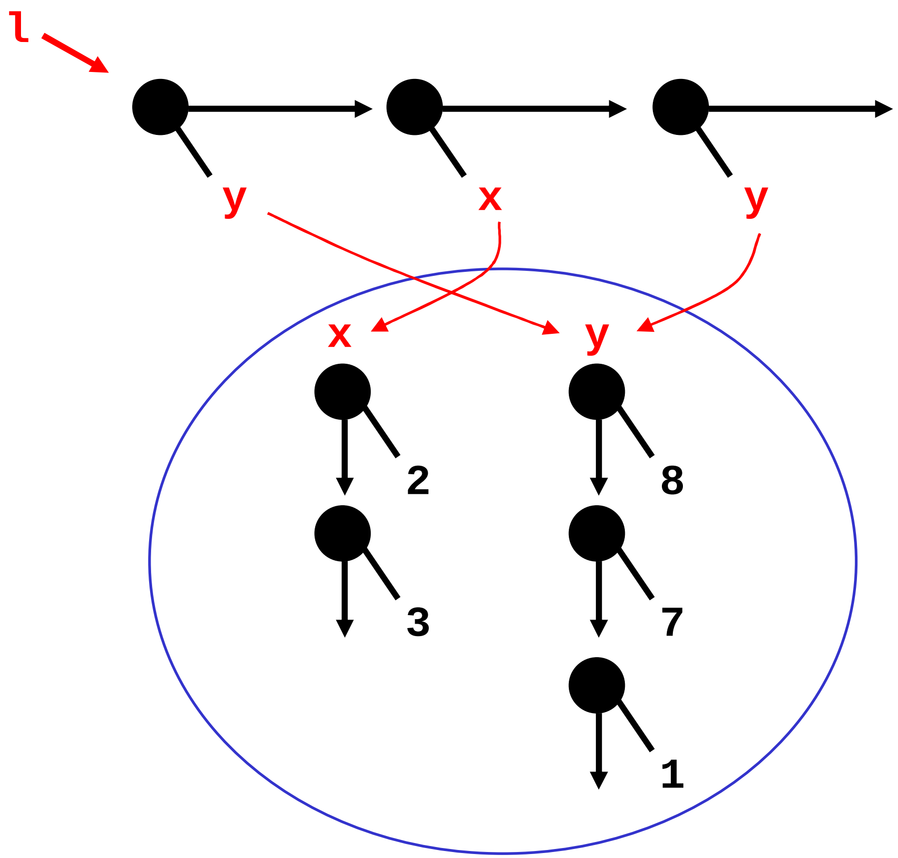
Reading the head of a mutable list:

```
Spec mlist_hd l |R>> ∀ x t,
  keep R (l ~> Mlist Id Id x t) (\= x).
```

Aliasing with groups

List of aliased lists of integers: l ~> Mlist Id_{loc} L

where L : list loc and L = y::x::y::nil



Group predicate:

Group (MList Id_{int}) M

where M : map loc (list int)

and M[x] = 2::3::nil

and M[y] = 8::7::1::nil

Operation on groups

Insertion and removal for groups:

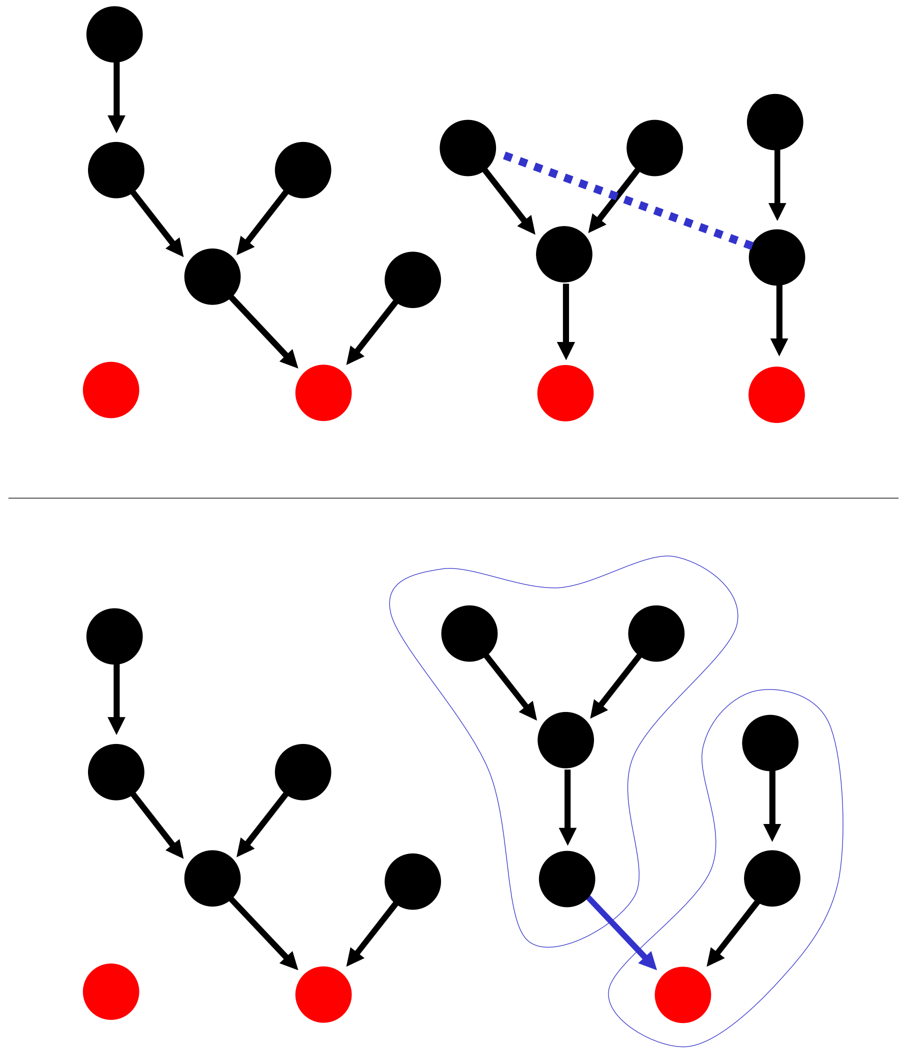
```
Lemma Group_add :
  Group T M \* (x ~> T X)
= Group T (M\(x:=X)).

Lemma Group_rem : x \indom M ->
  Group T M
= Group T (M \-- x) \* (x ~> T (M\(x))).
```

Derived operations for groups of references:

```
Spec get (l:loc) |R>>
  ∀(M:map loc A), l \indom M ->
  keep R (Group (Ref Id) M) (\= M\(l)).

Spec set (l:loc) (v:A) |R>>
  ∀(M:map loc A), l \indom M ->
  R (Group (Ref Id) M) (# Group (Ref Id) (M\(l:=v))).
```

Union-find data structure

```
type cell = content ref
and content =
  Node of cell | Root

let rec repr x =
  match !x with
  | Root -> x
  | Node y -> repr y

let create () =
   ref Root

let same x y =
  repr x == repr y

let union x y =
  let rx = repr x in
  let ry = repr y in
  if rx != ry
    then rx := Node ry
```



Union-find: invariants

```
Inductive is_repr (M:map loc content) : loc->loc->Prop:=
| is_repr_root : ∀ x,
   binds M x Root -> is_repr M x x
| is_repr_step : ∀ x y r,
   binds M x (Node y) -> is_repr M y r -> is_repr M x r.

Definition is_equiv M x y :=
  ∃r, is_repr M x r /\ is_repr M y r.

Definition is_forest M :=
  ∀x, x \indom M -> ∃r, is_repr M x r.

Definition UFgraph (G:graph loc) : Hprop :=
  Hexists M, Group (Ref Id) M \*
    [ is_forest M /\ dom M = nodes G
      /\ is_equiv M = connected G ].

Definition connected A (G:graph A) : A->A->Prop :=
  rstclosure (fun x y => (x,y) \in edges G).
```

Union-find: connected components

```
Spec repr x |R>> forall M,
  is_forest M -> x \indom M ->
  keep R (Group (Ref Id) M) (fun r => [is_repr M x r])


Spec create () |R>> forall G,
  R (UFgraph G) (fun r => [r \notin nodes G]
                \* UFgraph (add_node G r)).

Spec same x y |R>> forall G,
  x \in nodes G -> y \in nodes G ->
  keep R (UFgraph G) (\= istrue (connected G x y)).

Spec union x y |R>> forall G,
  x \in nodes G -> y \in nodes G ->
  R (UFgraph G) (# UFgraph (add_edge G x y)).
```

Union-find: partial equiv. relations

```
Definition UF (B:binary loc): Hprop :=
  Hexists M, Group (Ref Id) M \* [ per B /\
   is_forest M /\ dom M = per_dom B /\ is_equiv M = B ].

Definition per_dom A (B:binary A) := \set{ x | B x x}.

Definition add_single A (B:binary A) (x:A) (y:A) :=
  stclosure (fun u v => B u v \/ (u=x /\ v=y)).

Spec create () |R>> forall B,
  R (UF B) (fun r => [r \notin per_dom B]
                \* UF (add_single B r r)).

Spec same x y |R>> forall B,
  x \in per_dom B -> y \in per_dom B ->
  keep R (UF B) (\= istrue (B x y)).

Spec union x y |R>> forall B,
  x \in per_dom B -> y \in per_dom B ->
  R (UF B) (# UF (add_single B x y)).
```

Union-find: verification

– Lemmas: 140 lines in 17 lemmas (14 inductions)
– Verification: 34 lines, invoking those lemmas

```
Lemma union_spec :
  Spec union x y |R>> forall B,
    x \in per_dom B -> y \in per_dom B ->
    R (UF B) (# UF (add_edge B x y)).
Proof.
  xcf. introv Dx Dy. unfold UF. xextract as M (PM&FM&DM&EM).
  rewrite <- DM in *. xapp*. intros Rx. xapp*. intros Ry. xapps. xif.
  (* case [rx <> ry] *)
  xapp*. apply* is_repr_in_dom_r. hsimpl. splits.
    applys* per_add_edge.
    apply* is_forest_add_edge; apply* is_repr_binds_root.
    rewrite per_dom_add_edge. rewrite <- DM.
     rewrite* dom_update_in. set_eq*. forwards*: is_repr_binds_root Rx.
    apply* inv_add_edge.
  (* case [rx = ry] *)
  xrets. splits*.
    applys* per_add_edge.
    rewrite per_dom_add_edge. rewrite <- DM. set_eq*.
    rewrite* add_edge_already. rewrite* <- EM.
Qed.
```

Composition function

```
> let compose g f x =
>   g (f x)
```

The behavior of "compose g f x" is the same as that of "g (f x)", so "compose g f x" admits pre H and post Q if the characteristic formula of "g (f x)" holds of H and Q.

```
Lemma compose_spec : forall A B C,
  Spec compose (g:Func) (f:Func) (x:A) |R>>
    ∀(H:hprop)(Q:C->hprop),
    (Let y := App f x; in App g y;) H Q -> R H Q.
```

On the goal "(App compose g f x;) H Q", the tactic xapp produces "(Let y := App f x; in App g y;) H Q", just as if the code of compose had been inlined.

Counter function

```
> let make_counter () =
>   let r = ref 0 in
>   let f () = incr r; !r in
>   f

Definition CounterSpec I f :=
  Spec f () |R>> ∀m,
    R (I m) (\=(m+1) \*+ (I (m+1))).

Definition Counter (n:int) (f:func) : hprop :=
  Hexists (I:int->hprop), I n \* [CounterSpec I f].

Lemma make_counter_spec :
  Spec make_counter () |R>> R [] (~> Counter 0).
Proof.
  xcf. xapps. sets I: (fun n:int => r ~~> n).
  xfun (CounterSpec I). xgo*. unfold I. hsimpl.
  xret. hdata_simpl Counter. hsimpl~ I.
Qed.
```

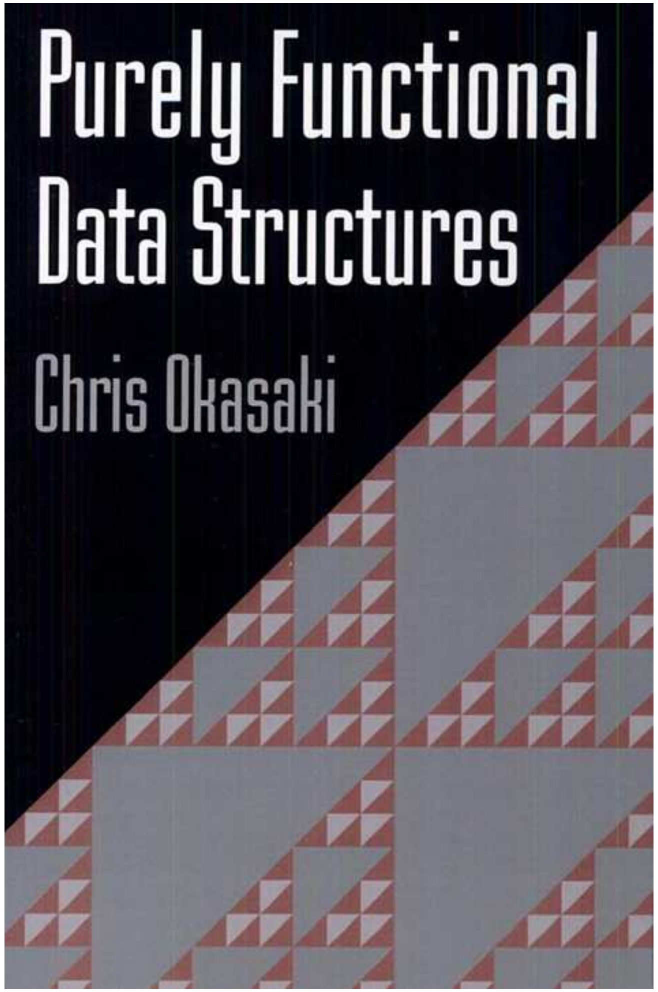
Calls to counter functions

```
Lemma Counter_apply : ∀ (f:Func) (n:int),
  (App f tt;) (f ~> Counter n)
              (\= (n+1) \*+ f ~> Counter (n+1)).


> let step_all (l:(unit->int)list) =
>   List.iter (fun f -> ignore (f())) l

Lemma step_all_spec :
  Spec step_all (l:list Func) |R>> ∀(L:list int),
    R (l ~> List Counter L)
      (# l ~> List Counter (map (fun i => i+1) L)).
```

Specification of List.iter

```
Spec iter (f:Func) (l0:list A) | R>>
  ∀ (H:Hprop) (Q:unit->Hprop),
   (∀ (S:list A->Hprop->(unit->Hprop)->Prop),
     is_local_1 S ->
     (∀ l H' Q',
        match l with
        | nil => (Ret tt) H' Q'
        | x::l' => ((App f x;) ;; S l') H' Q'
        end -> S l H' Q') ->
     S l0 H Q) ->
  R H Q.
```

Purely functional data structures



Formalized in particular:

– red-black trees

– lazy queues

– realtime queues,

– bootstrapped queues

– splay heaps

– binomial heaps

– leftist heaps

– pairing heaps

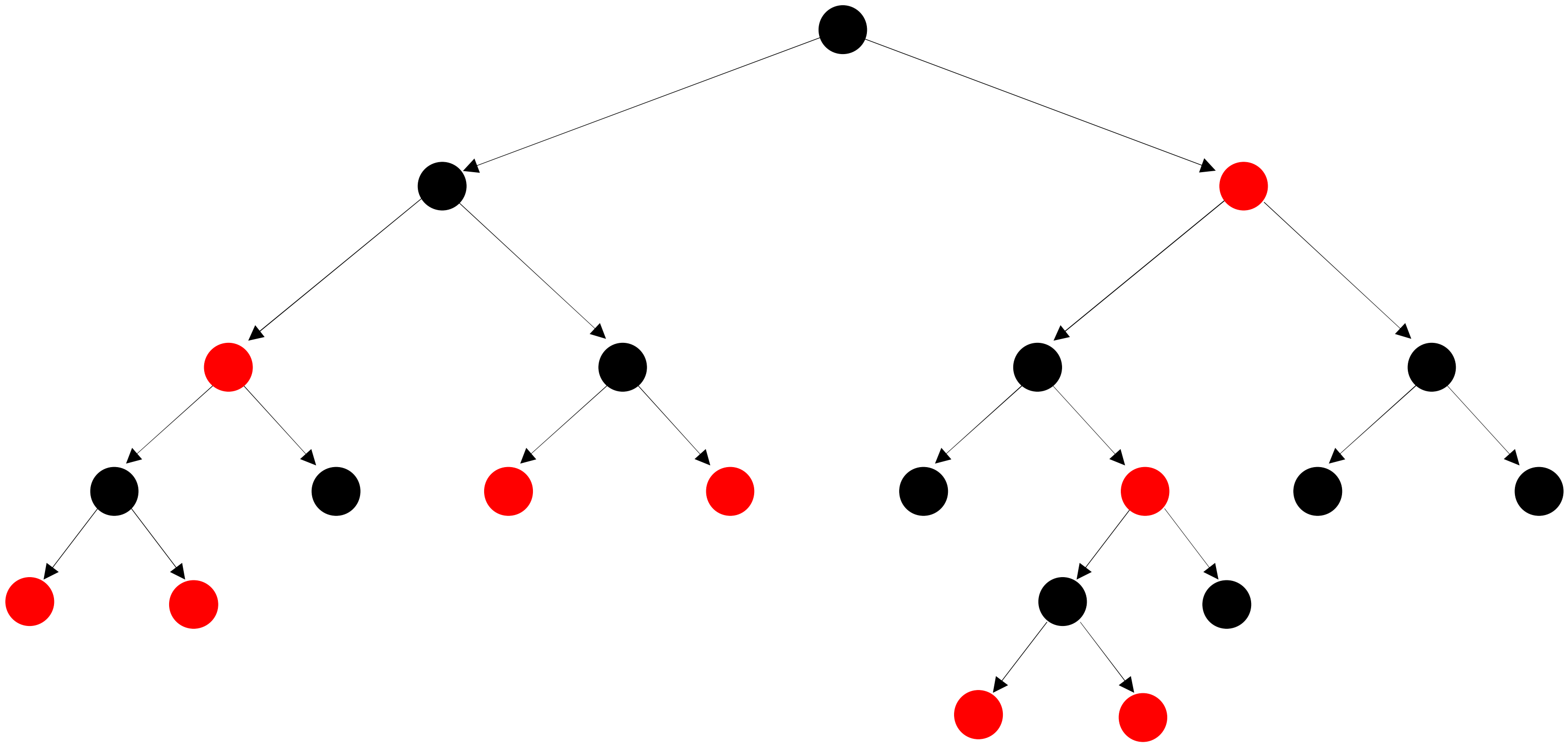
– concatenable lists

– binary random access lists

Code

```
module RedBlackSet (Element : Ordered) : Fset = struct
 type color = Red | Black
 type tree = Empty | Node of color * tree * elem * tree

 let rec member x = function
 | Empty -> false
 | Node (_,a,y,b) -> if Element.lt x y then member x a
                     else if Element.lt y x then member x b else true

 let balance = function
 | (Black, Node (Red, Node (Red, a, x, b), y, c), z, d)
 | (Black, Node (Red, a, x, Node (Red, b, y, c)), z, d)
 | (Black, a, x, Node (Red, Node (Red, b, y, c), z, d))
 | (Black, a, x, Node (Red, b, y, Node (Red, c, z, d))) ->
     Node (Red, Node (Black, a, x, b), y, Node (Black, c, z, d))
 | (c,a,x,y) -> Node (c,a,x,y)

 let rec insert x s =
   let rec ins = function
     | Empty -> Node (Red, Empty, x, Empty)
     | Node (col, a, y, b) as s ->
        if Element.lt x y then balance (col, ins a, y, b)
        else if Element.lt y x then balance (col, a, y, ins b)
        else s in
  let Node (_, a, y, b) = ins s in Node (Black, a, y, b)
```

Invariants of red-black trees



Binary search tree where:

1) Same number of black nodes in every path
2) No red node has a red child
3) Root is always black

Invariant for red-black trees

The invariant "**inv n t E**" holds if the tree **t** represents the set **E** and every path in **t** contains **n** black nodes.

```
Inductive inv : nat -> tree t -> set T -> Prop :=
 | inv_empty : forall rok,
     inv 0 Empty \{}
 | inv_node : forall rok n m col a x b A X B,
     inv m a A -> inv m b B -> rep x X ->
     foreach (is_lt X) A -> foreach (is_gt X) B ->
     (n = match col with Black => m+1 | Red => m end) ->
     (match col with | Black => True
                     | Red => node_color a = Black
                           /\ node_color b = Black end) ->
     inv n (Node col a x b) (\{X} \u A \u B).

Instance tree_rep : Rep (tree t) (set T) := { rep :=
 fun e E => exists n, inv n e E /\ node_color e = Black }.
```

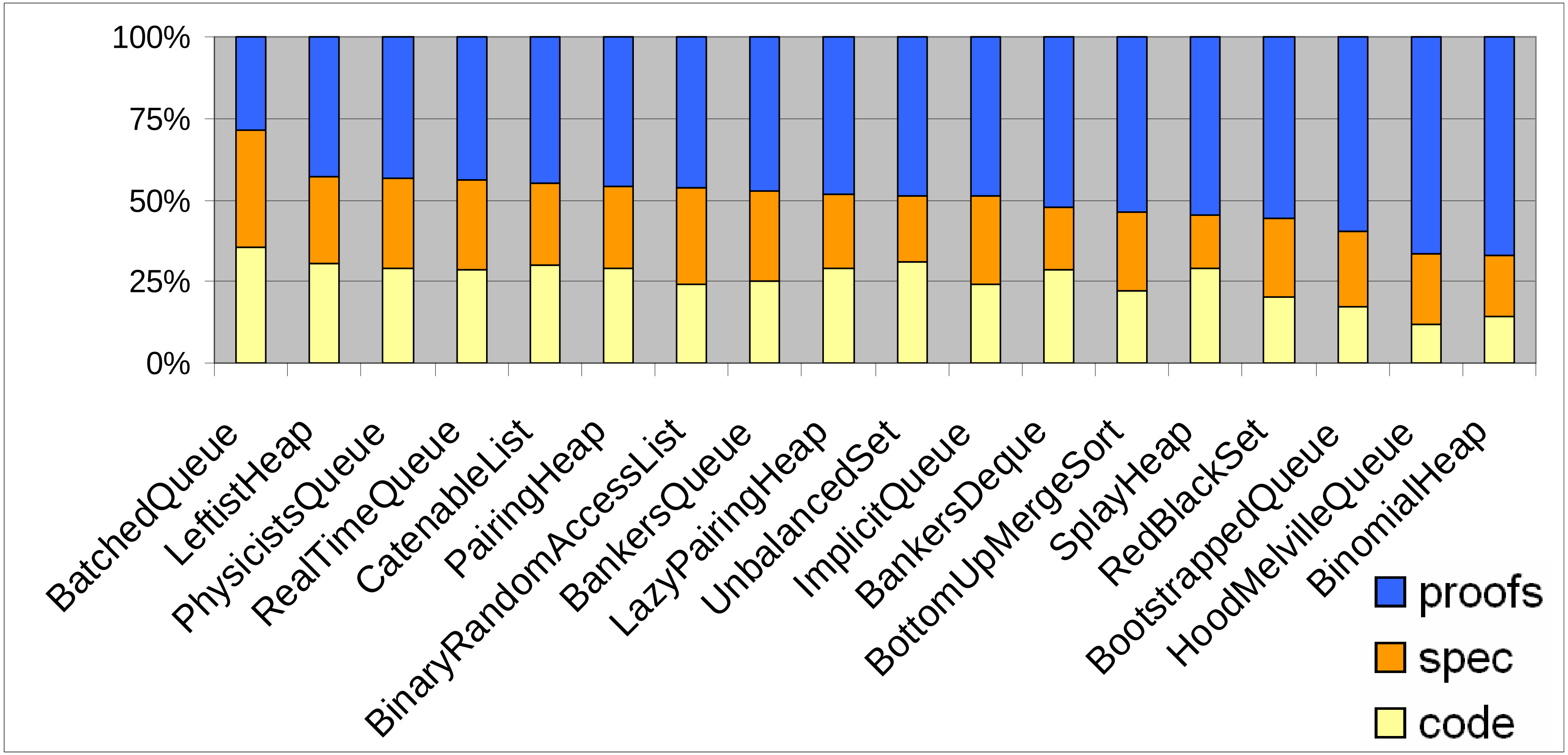
Obligation

```
(x : elem) (X : OS.T) (ins : val) (n : nat) (col : color)
(a : tree) (y : elem) (b : tree) (A : set T) (Y : T)
(B : set T) (m : nat) (_a : tree) (s : tree)
InvA : inv m a A
InvB : inv m b B
RepX : rep x X
RepY : rep y Y
GtY : foreach (is_lt Y) A
LtY : foreach (is_gt Y) B
Col : match col with
      | Red => node_color a = Black /\ node_color b = Black
      | Black => True end
Num : n = match col with Red => m | Black => S m end
Es : s = Node col a y b
C1 : X < Y
P_a : inv m _a ('{X} \u A)
___________________________________________________________

(App balance (col, _a, y, b);)
  (fun e' => inv n e' ('{X} \u '{Y} \u A \u B))
```

Proof

```
Lemma insert_spec : RepTotal insert (X;elem) (E;set) |R>>
  R [] (= \{X} \u E ;; set).
Proof.
  xcf. introv RepX (n&InvE&HeB).
  xfun_induction_nointro (ins_spec X) size.
    clears s n E. intros e IH n E InvE. inverts InvE as.
    xgo*. simpl. constructors~.
    introv InvA InvB RepY GtY LtY Col Num. xgo~.
    (* -- case insert left -- *)
    destruct~ col; destruct (node_color a); tryifalse; auto.
    ximpl as e. simpl. applys_eq~ Hx 1 3.
    (* -- case insert right -- *)
    destruct~ col; destruct (node_color b); tryifalse; auto.
    ximpl as e. simpl. applys_eq~ Hx 1 3.
    (* -- case no insertion -- *)
    asserts_rewrite~ (X = Y). apply~ nlt_nslt_to_eq.
    subst s. simpl. destruct col; constructors~.
  xlet. xapp~. inverts P_x5; xgo. fset_inv. exists~.
Qed.
```

Statistics Okasaki's book



Total: **564** non-blank lines of **Caml** (very concise code)

2489 non-blank lines of **Coq** (2min. to compile)

Not presented in this talk

- General specifications for the swap function
- Landin's knot (recursion through the store)
- Sparse arrays (first task from VACID-0 challenge)
- Bytecode compiler and interpretor for mini-ML

Additional features

Implemented but not discussed:

– Curried n-ary functions

– Pattern matching

– Mutual recursion

– Polymorphism, polymorphic recursion

– Null pointers and strong updates

– Modules and functors

Not supported:

– modulo arithmetics, real numbers (seems hard)

– catchable exceptions (future work)

– object-oriented programming (future work)

– nontrivial uses of laziness (future work)

Verification Condition Generator

Idea:

– annotate programs with specifications and invariants

– extract proof obligations that entail correctness

– discharge those obligations using SMT solvers

Difficulties:

– hard to progress when SMT solvers fail

– hard to even read proof obligations

CF:

– support interactive proofs with immediate feedback

– still some opportunity for automated reasoning

Deep embedding

Idea:

– axiomatize the syntax and semantics

– state theorem "such program admits such behavior"

Difficulties:

– explicit substitution of program variables

– no identification between program and logical values

CF:

$\rightarrow$ an abstract layer built on top of a deep embedding

– still identify program and logical variables and values

Shallow embedding

Idea:

– write programs in the logic of a proof assistant

– extract conventional purely-functional code

Difficulties:

– logical functions must be total

– side-effects must be encoded in a monad

– can be hard to recognize ghost variables

CF:

– no constraints on the programming language

– identify program and logical values except functions

Characteristic formulae

Originates in process calculi:

– a temporal-logic formula describes a process

– two processes are behavioraly equivalent if and only if their formulae are logically equivalent

– "characteristic" formula generation is automatable

Honda, Berger and Yoshida:

– applied the idea to build program logics in which sound and complete formulae can be expressed

– described an algorithm for building weakest pre- and stronger post-conditions

CF:

– target a standard logic, so leads to an effective tool

– exploit the strength of higher-order logic

Soundness and completeness

Soundness theorem: ⌊V⌋ takes Coq values into Caml

$$\left\{\begin{array}{l} \vdash\ t\ :\ T \\ [\![t]\!]\ H\ Q \\ H\ h_i \\ h_i \perp h_k \end{array}\right. \Rightarrow \exists V\ h_f\ h_g. \left\{\begin{array}{l} \vdash\ \lfloor V \rfloor\ :\ T \\ t_{/\lfloor h_i \rfloor + \lfloor h_k \rfloor} \Downarrow \lfloor V \rfloor_{/\lfloor h_f \rfloor + \lfloor h_k \rfloor + \lfloor h_g \rfloor} \\ h_f \perp h_k \perp h_g \\ Q \lfloor V \rfloor h_f \end{array}\right.$$

Completeness theorem: (slightly simplified)

$$\left\{\begin{array}{l} \vdash\ t\ :\ T \\ t_{/m} \Downarrow v_{/m'} \end{array}\right. \Rightarrow \quad [\![t]\!]\ (\mathsf{mgh}\ m)\ (\mathsf{mgp}\ v\ m')$$

Completeness, special case:

$$t_{/\emptyset} \Downarrow n_{/m} \quad \Rightarrow \quad [\![t]\!]\ [\,]\ (\lambda x.\,[x = n])$$

Formalization in Coq

For the language IMP (assign, while, skip, if, seq)

```
(* Axiomatization of source language *)
Definition heap := nat -> nat.
Inductive command := ...
Inductive eval : command -> heap -> heap -> Prop := ..

(* Construction of characteristic formulae *)
Definition Hprop := heap -> Prop.
Definition Formula := Hprop -> Hprop -> Prop.
Fixpoint cf (c : command) : Formula := ...

(* Soundness and completeness results *)
Theorem cf_sound : ∀ (c:command) (P Q:Hprop) (h:heap),
  cf c P Q -> P h -> ∃ h', eval c h h' /\ Q h'.
Theorem cf_complete : ∀ (c:command) (h h':state),
  eval c h h' -> cf c (= h) (= h').
```

Towards a fully-verified chain

Specification

↓ Mechanically-verified proofs

Characteristic formula

Soundneness proof on paper ↓ Mechanically-verified characteristic formulae

Source code

OCaml compiler ↓ Mechanically-verified compiler

Machine code

Thanks!

Further information on characteristic formulae:

Characteristic Formulae for Mechanized Program Verification

Arthur Charguéraud, September 2010

http://arthur.chargueraud.org/research/2010/thesis