Interactive Verification of Imperative Programs

Arthur Charguéraud
Inria

May 2015

Program verification

Goal: relate a program with a formal specification.

```
let sieve n = (* Erathostenes *)
  let t = Array.make n true in
  t.(0) <- false;
  t.(1) <- false;
  let i = ref 2 in
  while !i < n do
    if t.(!i) then begin
      let r = ref !i in
      while !r * !i < n do
        t.(!r * !i) <- false;
        incr r;
      done;
    end;
    incr i;
  done;
  t
```

Pre-condition:

$$n > 1$$

Post-condition: produces `t` such that

$$\mathtt{length}\ t = n \wedge \forall i,\ 0 \leqslant i < n \rightarrow t[i] = \mathtt{true} \leftrightarrow \mathtt{prime}\ i$$

Where:

```
Definition prime i :=
  i > 1 ∧ ∀d r, 1 < d < i → i ≠ d * r.
```

Contents

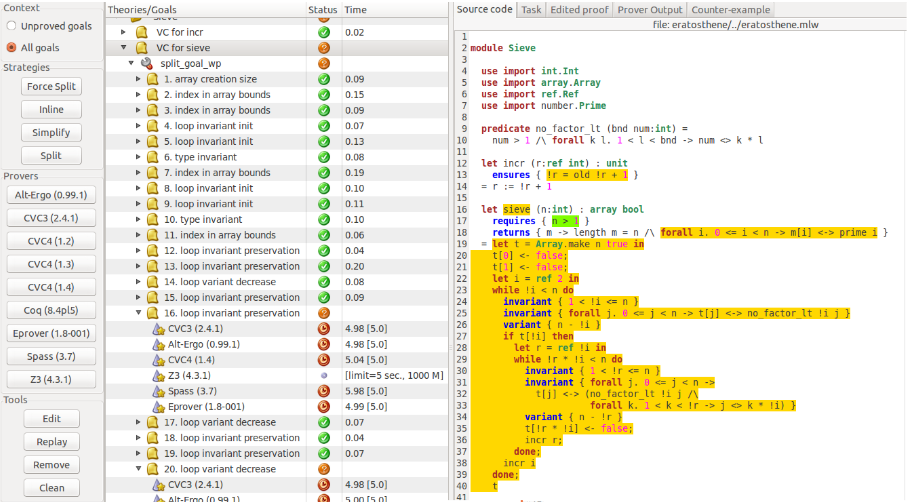
Contents

Annotated program

```
let sieve n
  requires { n > 1 }
  returns { t -> length t = n /\ (forall i. 0 <= i < n -> t[i] <-> prime i) }
= let t = Array.make n true in t[0] <- false; t[1] <- false;
  let i = ref 2 in
  while !i < n do
    invariant { 1 < !i <= n }
    invariant { forall j. 0 <= j < n -> t[j] <-> no_factor_lt !i j }
    variant { n - !i }
    if t[!i] then begin
      let r = ref !i in
      while !r * !i < n do
        invariant { 1 < !r <= n }
        invariant { forall j. 0 <= j < n ->
          t[j] <-> (no_factor_lt !i j /\ forall k. 1 < k < !r -> j <> k * !i) }
        variant { n - !r }
        t[!r * !i] <- false;
        incr r;
      done; end; incr i
  done; t

predicate no_factor_lt i j =
  j > 1 /\ forall k l. 1 < l < i /\ k > 1 -> j <> k * l
```

The Why3 interface



Weakest-precondition

13. loop invariant preservation	✓	0.15
14. loop variant decrease	✓	0.02
15. loop invariant preservation	✓	0.02
16. loop invariant preservation	?	
Alt-Ergo (0.99.1)		5.00 [5.0]
CVC3 (2.4.1)		4.97 [5.0]
CVC4 (1.4)		5.05 [5.0]
Z3 (4.3.1)		4.98 [5.0]
Spass (3.7)		5.31 [5.0]
Eprover (1.8-001)		4.98 [5.0]
17. loop variant decrease	✓	0.02
18. loop invariant preservation	✓	0.02
19. loop invariant preservation	?	
Alt-Ergo (0.99.1)		4.97 [5.0]
CVC3 (2.4.1)		4.98 [5.0]
CVC4 (1.4)		5.99 [5.0]
Z3 (4.3.1)		4.98 [5.0]
Spass (3.7)		4.98 [5.0]
Eprover (1.8-001)		4.97 [5.0]
20. loop variant decrease	✓	0.03
21. type invariant	✓	0.03
22. postcondition	?	
Alt-Ergo (0.99.1)		4.91 [5.0]
CVC3 (2.4.1)		4.98 [5.0]
CVC4 (1.4)		5.98 [5.0]
Z3 (4.3.1)		4.91 [5.0]
Spass (3.7)		5.29 [5.0]
Eprover (1.8-001)		4.98 [5.0]

```
predicate no_factor_lt (bnd:int) (num:int) =
  num > 1 /\ (forall k:int, l:int. 1 < l /\ l < bnd -> not num = (k * l))

goal WP_parameter_sieve :
  forall n:int.
   n > 1 ->
    n >= 0 ->
     0 <= n ->
      0 <= 0 /\ 0 < n ->
       (forall t:map int bool.
         0 <= n && t = set (const True:map int bool) 0 False ->
          0 <= 1 /\ 1 < n ->
           (forall t1:map int bool.
             0 <= n && t1 = set t 1 False ->
              (forall i:int, t2:map int bool.
                (1 < i /\ i <= n) /\
                 (forall j:int.
                   0 <= j /\ j < n -> get t2 j = True <-> no_factor_lt i j) ->
                 i < n ->
                  0 <= n && 0 <= i /\ i < n ->
                   get t2 i = True ->
                    (forall r:int, t3:map int bool.
                      (1 < r /\ r <= n) /\
                       (forall j:int.
                         0 <= j /\ j < n ->
                          get t3 j = True <->
                           no_factor_lt i j /\
                            (forall k:int.
                              1 < k /\ k < r -> not j = (k * i))) ->
                      not (r * i) < n ->
                       (forall i1:int.
                         i1 = (i + 1) ->
                          (forall j:int.
                            0 <= j /\ j < n ->
                             get t3 j = True <-> no_factor_lt i1 j))))))
end
```

User intervention using assertions

Theories/Goals	Status	Time
eratosthene.mlw	✓	4.35
Sieve	✓	4.35
VC for incr	✓	0.01
VC for sieve	✓	4.34
split_goal_wp	✓	4.34
1. array creation size	✓	0.02
2. index in array bounds	✓	0.02
3. index in array bounds	✓	0.01
4. loop invariant init	✓	0.02
5. loop invariant init	✓	0.02
6. type invariant	✓	0.03
7. index in array bounds	✓	0.01
8. loop invariant init	✓	0.02
9. loop invariant init	✓	0.05
10. type invariant	✓	0.02
11. index in array bounds	✓	0.02
12. loop invariant preservation	✓	0.01
13. loop invariant preservation	✓	0.06
14. loop variant decrease	✓	0.04
15. assertion	✓	0.15
16. loop invariant preservation	✓	0.02
17. loop invariant preservation	✓	0.15
18. loop variant decrease	✓	0.03
19. assertion	✓	0.30
20. loop invariant preservation	✓	0.02
21. loop invariant preservation	✓	0.04
22. loop variant decrease	✓	0.03

Source code | Task | Edited proof | Prover Output | Counter-example

file: eratosthene/../eratosthene.mlw

```
let sieve (n:int) : array bool
  requires { n > 1 }
  returns { m -> length m = n /\ forall i. 0 <= i < n -> m[i] <-> prime i }
= let t = Array.make n true in
  t[0] <- false;
  t[1] <- false;
  let i = ref 2 in
  while !i < n do
    invariant { 1 < !i <= n }
    invariant { forall j. 0 <= j < n -> t[j] <-> no_factor_lt !i j }
    variant { n - !i }
    if t[!i] then begin
      let r = ref !i in
      while !r * !i < n do
        invariant { 1 < !r <= n }
        invariant { forall j. 0 <= j < n ->
          t[j] <-> (no_factor_lt !i j /\
                  forall k. 1 < k < !r -> j <> k * !i) }
        variant { n - !r }
        t[!r * !i] <- false;
        incr r;
      done;
      assert { forall j. 0 <= j < n /\ t[j] ->
        (forall k l. 1 < l < !i + 1 -> j = k * l /\ k > 1 ->
          (if l = !i then k < !r && false else false) && false) &&
        no_factor_lt (!i+1) j }
    end else assert { forall j. 0 <= j < n /\ no_factor_lt !i j ->
      (forall k l. 1 < l < !i + 1 -> j = k * l /\ k > 1 ->
        (if l = !i then (forall k0 l. 1 < l < !i /\ k0 > 1 /\ !i = k0 * l ->
            j = (k*k0) * l && false) && false
                   else false) && false) && no_factor_lt (!i+1) j };
    incr i
  done;
  t
```

User intervention using Coq

CoqIde

coq.v

```
(* Why3 assumption *)
Definition no_factor_lt (bnd:Z) (num:Z): Prop := (1%Z < num)%Z /\
  forall (k:Z) (l:Z), (((1%Z < l)%Z /\ (l < bnd)%Z) /\ (1%Z < k)%Z) ->
  ~ (num = (k * l)%Z).

(* Why3 goal *)
Theorem WP_parameter_sieve : forall (n:Z), (1%Z < n)%Z -> ((0%Z <= n)%Z ->
  ((0%Z <= n)%Z -> (((0%Z <= 0%Z)%Z /\ (0%Z < n)%Z) -> forall (t:(map.Map.map
  Z bool)), ((0%Z <= n)%Z /\
  (t = (map.Map.set (map.Map.const true: (map.Map.map Z bool)) 0%Z
  false))) -> (((0%Z <= 1%Z)%Z /\ (1%Z < n)%Z) -> forall (t1:(map.Map.map Z
  bool)), ((0%Z <= n)%Z /\ (t1 = (map.Map.set t 1%Z false))) -> forall (i:Z)
  (t2:(map.Map.map Z bool)), (((1%Z < i)%Z /\ (i <= n)%Z) /\ forall (j:Z),
  ((0%Z <= j)%Z /\ (j < n)%Z) -> (((map.Map.get t2 j) = true) <->
  (no_factor_lt i j))) -> ((i < n)%Z -> (((0%Z <= n)%Z /\ ((0%Z <= i)%Z /\
  (i < n)%Z)) -> ((~ ((map.Map.get t2 i) = true)) -> ((forall (j:Z),
  (((0%Z <= j)%Z /\ (j < n)%Z) /\ (no_factor_lt i j)) -> ((forall (k:Z)
  (l:Z), ((1%Z < l)%Z /\ (l < (i + 1%Z)%Z)%Z) -> (((j = (k * l)%Z) /\
  (1%Z < k)%Z) -> (((l = i) /\ ((forall (k0:Z) (l1:Z), (((1%Z < l1)%Z /\
  (l1 < i)%Z) /\ ((1%Z < k0)%Z /\ (i = (k0 * l1)%Z))) ->
  ((j = ((k * k0)%Z * l1)%Z) /\ False)) /\ False)) /\ False))) /\
  (no_factor_lt (i + 1%Z)%Z j))) -> forall (i1:Z), (i1 = (i + 1%Z)%Z) ->
  forall (j:Z), ((0%Z <= j)%Z /\ (j < n)%Z) -> (((map.Map.get t2
  j) = true) -> (no_factor_lt i1 j)))))))))).
Proof.
intros n h1 h2 h3 (h4,h5) t (h6,h7) (h8,h9) t1 (h10,h11) i t2 ((h12,h13),h14)
h15 (h16,(h17,h18)) h19 h20 i1 h21 j (h22,h23) h24.
Require Import Why3.
Ltac ae := why3 "Alt-Ergo,0.99.1,".
clear h20. rewrite h21 in *;clear h21. unfold no_factor_lt.
split. ae. intros. apply h14 in h24;[|ae].
destruct (Z_eq_dec l i);[|apply h24; ae].
rewrite e in *;clear e.
assert(exists k0 l0, (k0 * l0 = i /\ 1 < l0 < i /\ k0 > 1)%Z) by ae.
destruct H0 as [k0 [l0 [H1 [H2 H3]]]]. rewrite <- H1.
replace (k*(k0*l0))%Z with ((k*k0)*l0)%Z by ring. ae.
Qed.
```

Ready | Line: 90 Char: 7 | CoqIde started

Contents

Definition of pure programs in Coq: an example

Balanced binary search trees in Coq:

```
Inductive tree : Type :=
| Leaf : tree
| Node : info → tree → X.t → tree → tree.

Fixpoint union s1 s2 :=
 match s1, s2 with
  | Leaf, _ ⇒ s2
  | _, Leaf ⇒ s1
  | Node _ l1 x1 r1, _ ⇒
     let (l2',_,r2') := split x1 s2 in
     join (union l1 l2') x1 (union r1 r2')
 end.
```

Extracted OCaml code:

```
type tree =
| Leaf
| Node of info * tree * X.t * tree

let rec union s1 s2 =
  match s1 with
  | Leaf -> s2
  | Node (t0, l1, x1, r1) ->
    (match s2 with
     | Leaf -> s1
     | Node (t1, t2, t3, t4) ->
       let { t_left = l2';
             t_in = x;
             t_right = r2' }
          = split x1 s2 in
       join (union l1 l2') x1
            (union r1 r2'))
```

Interactive verification of imperative programs

Specification triple:

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

Interpretation (in total correctness):

$$\{H\}\ t\ \{Q\} \quad \equiv \quad \forall m.\ H\,m \ \Rightarrow\ \exists v m'. \quad t_{/m} \Downarrow v_{/m'} \ \wedge\ Q\,v\,m'$$

How to reason about triples $\{H\}\ t\ \{Q\}$ in an interactive theorem prover?

Solution 1: dynamic logics

- Use a *dynamic logic*, where $\langle t \rangle$ is a primitive construction such that:

$$\{H\}\ t\ \{Q\} \qquad \text{described as} \qquad (H \Rightarrow \langle t \rangle Q)$$

- Implemented in the *KeY* tool, for verifying Java programs.
- Requires a dedicated theorem prover.

Solution 2: monadic translation

- Encode an imperative t of type A as a pure monadic program t':

$$t' \;:\; \mathsf{Heap} \times \mathbb{N} \;\to\; \mathsf{Result}(\mathsf{Heap} \times A) + \mathsf{Timeout} + \mathsf{Error}$$

- $\{H\}\; t\; \{Q\}$ is equivalent to:

$$\forall m. \quad H\, m \quad \Rightarrow \quad \exists n v m'. \quad t'\,(m, n) = \mathsf{Result}(m', v) \;\wedge\; Q\, v\, m'$$

- Implemented in the *Ynot* tool, for verifying ML programs.
- Requires the monadic translation to fit the syntax of Coq.

Solution 3: characteristic formulae

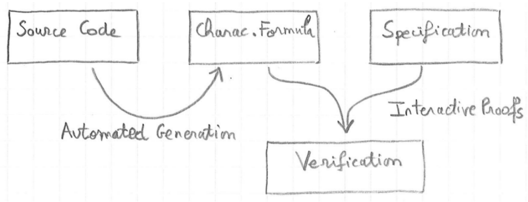
- The *characteristic formula* of a term t, written $[\![t]\!]$, is such that:

$$\forall H Q. \quad \{H\}\, t\, \{Q\} \;\Leftrightarrow\; [\![t]\!]\, H\, Q$$

- where $[\![t]\!]$ is a higher-order logic predicate (using $\forall, \exists, \wedge, \vee, \Rightarrow, \ldots$).
- $[\![t]\!] \;:\; (\text{Heap} \to \text{Prop}) \to (A \to \text{Heap} \to \text{Prop}) \to \text{Prop}.$
- Implemented in the *CFML* tool, for verifying Caml programs.

Contents

Characteristic formulae in practice



Five ingredients:

1. predicate transformers,
2. specific notation,
3. specific tactics,
4. the App predicate,
5. Separation Logic.

Characteristic formula for let bindings

Reasoning rule on triples:

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

Goal:

$$[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!]\,H\,Q \quad \Leftrightarrow \quad \{H\}\ (\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2)\ \{Q\}$$

Definition:

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

Notation for characteristic formulae

Characteristic formula:

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

Custom Coq notation:

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

We now have:

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

Tactics for characteristic formulae

Characteristic formula:

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

On a goal of the form "$\Gamma \vdash [\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!]\, H\, Q$", the tactic `xlet` produces two subgoals:

$$\Gamma \vdash [\![t_1]\!]\ H\ Q' \qquad \text{and} \qquad \Gamma, x \vdash [\![t_2]\!]\ (Q'\, x)\ Q$$

where Q' is a fresh unification variable (or is provided by the user).

Treatment of functions

Let "Func" be an abstract type used to represent effectful functions.
Let "App" be an abstract predicate with the following interpretation:

$$\mathsf{App}\, f\, v\, H\, Q \quad \Leftrightarrow \quad \{H\}\ (f\, v)\ \{Q\}$$

where:

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

Goal:

$$[\![f\, v]\!]\, H\, Q \quad \Leftrightarrow \quad \{H\}\ (f\, v)\ \{Q\}$$

Definition:

$$[\![f\, v]\!] \quad \equiv \quad \lambda H Q.\ \ \mathsf{App}\, f\, v\, H\, Q$$

Function definitions

Let f be a function defined as $\lambda x.\, t_1$. We provide the hypothesis:

$$\mathcal{P} \;\equiv\; (\forall x H' Q'.\; [\![t_1]\!]\, H'\, Q' \;\Rightarrow\; \mathsf{App}\, f\, x\, H'\, Q')$$

Definition:

$$[\![\mathsf{let}\, f = \lambda x.\, t_1 \,\mathsf{in}\, t_2]\!] \;\equiv\; \lambda H Q.\;\; \forall f.\; \mathcal{P} \;\Rightarrow\; [\![t_2]\!]\, H\, Q$$

Characteristic formulae for ML

$$
\begin{array}{lcl}
[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!] & \equiv & \lambda HQ.\;\; \exists Q'.\; [\![t_1]\!]\, H\, Q' \;\wedge\; \forall x.\; [\![t_2]\!]\, (Q'\, x)\, Q \\[1em]
[\![\mathsf{if}\, b \,\mathsf{then}\, t_1 \,\mathsf{else}\, t_2]\!] & \equiv & \lambda HQ.\;\; \quad (b = \mathsf{true} \Rightarrow [\![t_1]\!]\, H\, Q) \\
 & & \qquad\quad \wedge \;\; (b = \mathsf{false} \Rightarrow [\![t_2]\!]\, H\, Q) \\[1em]
[\![v]\!] & \equiv & \lambda HQ.\;\; H \rhd (Q\, v) \\
[\![f\, v]\!] & \equiv & \lambda HQ.\;\; \mathsf{App}\, f\, v\, H\, Q \\[1em]
[\![\mathsf{let}\, f = \lambda x.\, t_1 \,\mathsf{in}\, t_2]\!] & \equiv & \lambda HQ.\;\; \forall f.\; \mathcal{P} \;\Rightarrow\; [\![t_2]\!]\, H\, Q
\end{array}
$$

$$\text{where } \mathcal{P} \;\equiv\; (\forall x H' Q'.\; [\![t_1]\!]\, H'\, Q' \;\Rightarrow\; \mathsf{App}\, f\, x\, H'\, Q')$$

Characteristic formulae for ML, with notation

$$
\begin{array}{lcl}
[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!] & \equiv & \mathsf{Let}\ x = [\![t_1]\!]\ \mathsf{in}\ [\![t_2]\!] \\
[\![\mathsf{if}\, b \,\mathsf{then}\, t_1 \,\mathsf{else}\, t_2]\!] & \equiv & \mathsf{If}\ b\ \mathsf{then}\ [\![t_1]\!]\ \mathsf{else}\ [\![t_2]\!] \\
[\![v]\!] & \equiv & \mathsf{Ret}\ v \\
[\![f\ v]\!] & \equiv & \mathsf{App}\, f\, v \\
[\![\mathsf{let\,rec}\, f = \lambda x.\, t_1 \,\mathsf{in}\, t_2]\!] & \equiv & \mathsf{Let}\ f\ x\ =\ [\![t_1]\!]\ \mathsf{in}\ [\![t_2]\!]
\end{array}
$$

$[\![t]\!]$ is built compositionally, is of linear size, and is easy to read.
$[\![t]\!]$ describes the semantics of t in a correct and complete manner.

Contents

Separation Logic

Separation Logic: a technique that brings modularity in the specification and verification of programs with mutable state.

Introduced by Reynolds (2000 and 2002), with O'Hearn and Yang (2001), building on ideas from Burstall (1972).

Many adopters of Separation Logic

Micro-controller	Klein et al	NICTA
Assembly language	Chlipala et al	MIT
Operating system	Shao et al	Yale
C (drivers)	Yang et al	Oxford
C-light	Appel et al	Princeton
C11 (concurrent)	Vafeiadis, Parkinson et al	MPI and MSR
ML	Morisset et al	Harvard
Java	Parkinson et al	MSR and Cambridge
Java	Jacobs et al	Leuven
JavaScript	Gardner et al	Imperial College
Caml	Charguéraud	Inria
...	...	...

Separation Logic heap predicates

A heap predicate H has type "Heap $\rightarrow$ Prop", i.e. "$H\,m$" is a proposition.

In Separation Logic, heap predicates are obtained by composing:

$[\,]$	empty heap
$[P]$	empty heap with pure fact
$l \hookrightarrow v$	singleton heap
$\exists x.\, H$	existential quantification
$H \star H'$	separating conjunction

$$H \star H' \quad \equiv \quad \lambda m.\, \exists m_1 m_2.\, \begin{cases} m_1 \perp m_2 \\ m = m_1 \uplus m_2 \\ H_1\, m_1 \\ H_2\, m_2 \end{cases}$$

Examples of Separation Logic specifications

$$\{r \hookrightarrow 3\}\ (\texttt{incr r})\ \{\lambda().\ \ r \hookrightarrow 4\}$$

By the frame rule:

$$\{r \hookrightarrow 3 \star s \hookrightarrow 5\}\ (\texttt{incr r})\ \{\lambda().\ r \hookrightarrow 4 \star s \hookrightarrow 5\}$$

By the rule of consequence:

$$\{r \hookrightarrow 3 \star s \hookrightarrow 5\}\ (\texttt{incr r})\ \{\lambda().\ \ \exists n.\,[n > 3] \star (r \hookrightarrow n) \star (s \hookrightarrow 5)\}$$

Structural rules of Separation Logic

Frame rule:

$$\frac{\{H_1\}\ t\ \{\lambda x.\, H_1'\}}{\{H_1 \star H_2\}\ t\ \{\lambda x.\, H_1' \star H_2\}}$$

Rule of consequence:

$$\frac{H \rhd H' \qquad \{H'\}\ t\ \{Q'\} \qquad \forall x.\, Q'\,x \rhd Q\,x}{\{H\}\ t\ \{Q\}}$$

where: $H \rhd H' \;\equiv\; \forall m.\, H\,m \;\Rightarrow\; H'\,m.$

How to integrate these rules in characteristic formulae?

Integration of frame+consequence rule

$$
\begin{array}{lcl}
[\![\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \ \exists Q'.\ [\![t_1]\!]\,H\,Q' \ \wedge\ \forall x.\ [\![t_2]\!]\,(Q'\,x)\,Q) \\
[\![\mathsf{if}\,b\,\mathsf{then}\,t_1\,\mathsf{else}\,t_2]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \qquad\ (b = \mathsf{true} \Rightarrow [\![t_1]\!]\,H\,Q)\ ) \\
 & & \qquad\qquad\qquad\ \wedge\ \ (b = \mathsf{false} \Rightarrow [\![t_2]\!]\,H\,Q) \\
[\![v]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \ H \rhd Q\,v) \\
[\![f\ v]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \ \mathsf{App}\,f\,v\,H\,Q) \\
[\![\mathsf{let}\,f = \lambda x.\,t_1\,\mathsf{in}\,t_2]\!] & \equiv & \mathsf{local}\,(\lambda HQ.\ \ \forall f.\ \mathcal{P} \Rightarrow [\![t_2]\!]\,H\,Q)
\end{array}
$$

where:

$$
\mathsf{local}\,\mathcal{F} \ \equiv\ \lambda HQ.\ \ \exists H_1 H_2 Q_1.\left\{\begin{array}{l} H \rhd H_1 \star H_2 \\ \mathcal{F}\,H_1\,Q_1 \\ \forall x.\ \ Q_1\,x \star H_2 \rhd Q\,x \end{array}\right.
$$

Contents

Demo of CFML

Demo.

Contents

Time credits

Time credits:

$$\$\,n \quad : \quad \mathsf{Heap} \rightarrow \mathsf{Prop} \qquad\qquad \text{where } n \in \mathbb{Z}^+$$

Properties:

$$\$(n + n') \;=\; \$\,n \;\star\; \$\,n' \qquad \text{and} \qquad \$\,0 \;=\; [\,]$$

Complexity analysis

In a program execution:

Number of machine instructions $=$ O(Number of beta reductions).

Principle:

Force the spending of $\$1$ on every beta reduction.

Implementation: insert a call to a function "pay" at the head of every function body and every loop iteration, providing the specification:

$$\text{App}\,\text{pay}\,()\,(\$\,1)\,(\lambda().\,[\,])$$

Asymptotic analysis of the sieve

```
Theorem sieve_correct_and_fast :
  ∀n, n > 1 →
  (App sieve n;)
    ($ (cost n))
    (fun t ⇒ ∃M, t ⇝ Array M ⋆
      \[ length M = n ∧ ∀i, 0 ⩽ i < n → M[i] = isTrue (prime i)]).
```

where `cost` is $O(\log(\log n))$.

Independently formalized in Coq:

$$\sum_{\substack{1<p<n \\ p \text{ prime}}} \frac{n}{p} \quad = \quad O(n \log(\log n))$$

Potential in the sieve

For every prime p, we cross-out multiples of p that are smaller than n. There are at most $\frac{n}{p}$ such multiples. Hence the bound:

$$n + \sum_{\substack{1<p<n \\ p \text{ prime}}} \frac{n}{p}$$

```
while !i < n do
  pay();
  if t.(!i) then begin
    let r = ref !i in
    while !r * !i < n do
      pay();
      t.(!r * !i) <- false;
      incr r;
    done;
  end;
  incr i;
done
```

Potential for the outer loop at i:

$$\$ \left((n-i) + \sum_{\substack{i\leqslant p<n \\ p \text{ prime}}} \frac{n}{p} \right)$$

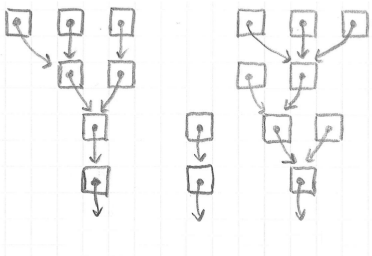
Potential for the inner loop at i, r:

$$\$ \left(\frac{n}{i} - r + 1 \right)$$

Amortized analysis

Time credits may be stored for later retrieval and consumption.

In Union-Find, with union-by-rank and path compression:



the amortized cost of union and find operations is $O(\alpha(n))$.

Contents

Interface for mutable sets

```
type 'a set

create : unit -> 'a set
is_empty : 'a set -> bool
search : 'a -> 'a set -> bool
insert : 'a -> 'a set -> unit
delete : 'a -> 'a set -> unit
```

Specification of mutable sets

Specification using a representation predicate: $t \leadsto \mathsf{Mset}\, E$.

$$\{[\,]\}\ (\texttt{create()})\ \{\lambda t.\ t \leadsto \mathsf{Mset}\, \varnothing\}$$

$$\{t \leadsto \mathsf{Mset}\, E\}\ (\texttt{is\_empty t})\ \{\lambda b.\ [b = \mathsf{isTrue}\,(E = \varnothing)] \star t \leadsto \mathsf{Mset}\, E\}$$

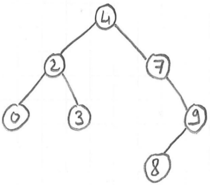
$$\{t \leadsto \mathsf{Mset}\, E\}\ (\texttt{search x t})\ \{\lambda b.\ [b = \mathsf{isTrue}\,(x \in E)] \star t \leadsto \mathsf{Mset}\, E\}$$

$$\{t \leadsto \mathsf{Mset}\, E\}\ (\texttt{insert x t})\ \{\lambda ().\ t \leadsto \mathsf{Mset}\,(E \cup \{x\})\}$$

$$\{t \leadsto \mathsf{Mset}\, E\}\ (\texttt{delete x t})\ \{\lambda ().\ t \leadsto \mathsf{Mset}\,(E \setminus \{x\})\}$$

where "$t \leadsto \mathsf{Mset}\, E$" is a notation of "$\mathsf{Mset}\, E\, t$".

Binary search trees




```
type node = contents ref
and contents = MLeaf | MNode of node * int * node

let rec search x t =
  match !t with
  | MLeaf -> false
  | MNode (t1, y, t2) ->
      if x < y then search x t1
      else if x > y then search x t2
      else true
```

Representation predicate for binary search trees

Definition, where t is a location, E is a set, and T is a pure binary tree:

$$t \rightsquigarrow \mathsf{Mset}\, E \quad \equiv \quad \exists T.\ t \rightsquigarrow \mathsf{Mtree}\, T \,\star\, [\mathsf{stree}\, T\, E]$$

For example, to prove:

$$\{t \rightsquigarrow \mathsf{Mset}\, E\}\ (\texttt{insert x t})\ \{\lambda().\ t \rightsquigarrow \mathsf{Mset}\,(E \cup \{x\})\}$$

we assume the existance of T such that:

$$t \rightsquigarrow \mathsf{Mtree}\, T \,\star\, [\mathsf{stree}\, T\, E]$$

and we need to exhibit a T' such that:

$$t \rightsquigarrow \mathsf{Mtree}\, T' \,\star\, [\mathsf{stree}\, T'\,(E \cup \{x\})]$$

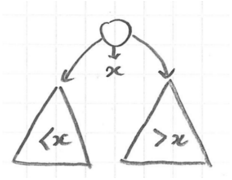
Specification of pure binary trees

Representation of pure trees in Coq:

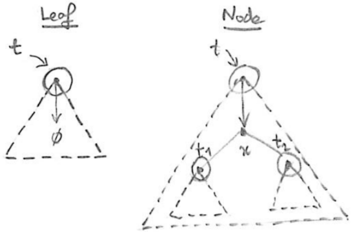
```
Inductive tree : Type :=
  | Leaf : tree
  | Node : tree → int → tree → tree.
```

Definition of $[\mathsf{stree}\, T\, E]$:

```
Inductive stree : tree → set int → Prop :=
  | stree_leaf :
      stree Leaf ∅
  | stree_node : ∀T1 x T2,
      stree T1 E1 →
      stree T2 E2 →
      foreach (is_lt x) E1 →
      foreach (is_gt x) E2 →
      stree (Node T1 x T2) ({x} ∪ E1 ∪ E2).
```

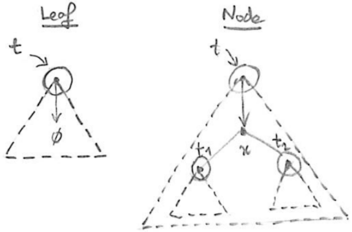


Binary trees in Separation Logic



$$
\begin{aligned}
t \rightsquigarrow \mathsf{Mtree}\, T \;\equiv\; \exists v.\; t \hookrightarrow v \;\star\; & \mathsf{match}\; T\; \mathsf{with} \\
& \mid \mathsf{Leaf} \Rightarrow [v = \mathsf{MLeaf}] \\
& \mid \mathsf{Node}\, T_1\, x\, T_2 \Rightarrow \exists t_1 t_2. \\
& \qquad\qquad [v = \mathsf{MNode}\, t_1\, x\, t_2] \\
& \qquad\qquad \star\, t_1 \rightsquigarrow \mathsf{Mtree}\, T_1 \\
& \qquad\qquad \star\, t_2 \rightsquigarrow \mathsf{Mtree}\, T_2
\end{aligned}
$$

Binary trees in Separation Logic



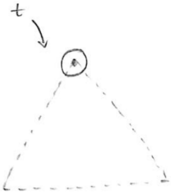
$$
\begin{aligned}
t \rightsquigarrow \mathsf{Mtree}\, T \;\; &\equiv \;\; \exists v.\ t \hookrightarrow v \;\star\;\; \mathsf{match}\ v\ \mathsf{with} \\
&\qquad\qquad\qquad\quad \mid \mathsf{MLeaf} \Rightarrow [T = \mathsf{Leaf}] \\
&\qquad\qquad\qquad\quad \mid \mathsf{MNode}\, t_1\, x\, t_2 \Rightarrow \exists T_1 T_2. \\
&\qquad\qquad\qquad\qquad\qquad [T = \mathsf{Node}\, T_1\, x\, T_2] \\
&\qquad\qquad\qquad\qquad\qquad \star\, t_1 \rightsquigarrow \mathsf{Mtree}\, T_1 \\
&\qquad\qquad\qquad\qquad\qquad \star\, t_2 \rightsquigarrow \mathsf{Mtree}\, T_2
\end{aligned}
$$

$$
t \rightsquigarrow \mathsf{Mtree}\, T \;\; \equiv \;\; \exists v.\ t \hookrightarrow v \;\star\; v \rightsquigarrow \mathsf{Contents}\, T
$$

Verification of search

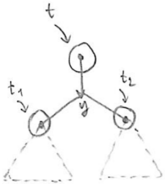
```
let rec search x t = match !t with
  | MLeaf -> false
  | MNode (t1, y, t2) -> if x < y then search x t1
                          else if x > y then search x t2
                          else true
```

Initial



$t \rightsquigarrow \mathsf{Mtree}\, T$

Focused

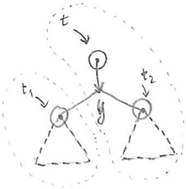


$t \hookrightarrow (\mathsf{MNode}\, t_1\, x\, t_2)$

$\star\ t_1 \rightsquigarrow \mathsf{Mtree}\, T_1$

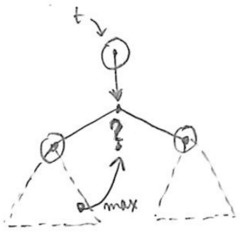
$\star\ t_2 \rightsquigarrow \mathsf{Mtree}\, T_2$

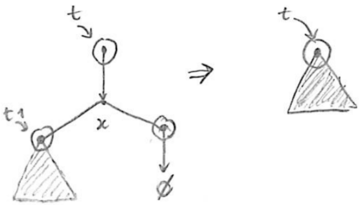
Framed



$t_1 \rightsquigarrow \mathsf{Mtree}\, T_1$

Implementation of delete






```
let rec delete x t =
  match !t with
  | MLeaf -> ()
  | MNode (t1, y, t2) ->
     if x < y then delete x t1
     else if x > y then delete x t2
     else match !t1 with
          | MLeaf -> t := !t2
          | _ -> let m = extract_max t1 in
                 t := MNode (t1, m, t2)
```

```
let rec extract_max t =
  match !t with
  | MLeaf -> assert false
  | MNode (t1, x, t2) ->
    match !t2 with
    | MLeaf -> t := !t1; x
    | _ -> extract_max t2
```

Demo: binary search trees

Demo.

Contents

Summary

Program verification using characteristic formulae:

- correctness and completeness,
- linear size, readable formulae,
- maintainable proof scripts,
- expressive, higher-order logic specifications,
- modularity of Separation Logic,
- support for amortized complexity analysis.

Not covered in this talk

- recursive ownership (e.g. arrays of arrays),
- sharing (e.g. union-find cells),
- higher-order functions (e.g. fold),
- advanced used of the frame rules.

→ For details: MPRI "Proof of programs" course notes (2015).

Future work

On-going projects:

- big-O notation for asymptotics,
- smooth integration of SMT provers,

Future projects:

- support for exceptions,
- characteristic formulae for other programming languages,
- realization of the characteristic formulae axioms.

Thanks!

Pointers:

- Characteristic Formulae for the Verification of Imperative Programs (HOSC 2012)
- Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation (ITP 2015, with F. Pottier)
- MPRI "Proof of programs" Course Notes (2015)
- Upcoming EPIT Coq spring school: `http://www.epit2015.website/`
- `http://arthur.chargueraud.org/softs/cfml`