Towards a programming language that makes verification easier

Arthur Charguéraud

INRIA

November 2012

Motivation

Goal: produce code that is correct <u>and</u> efficient

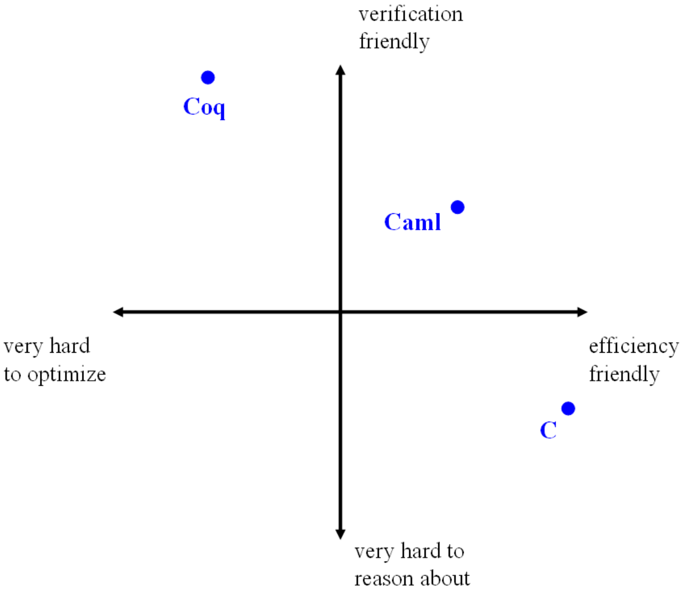
Correct code:

- mechanically-verified
- or just more likely to be bug-free

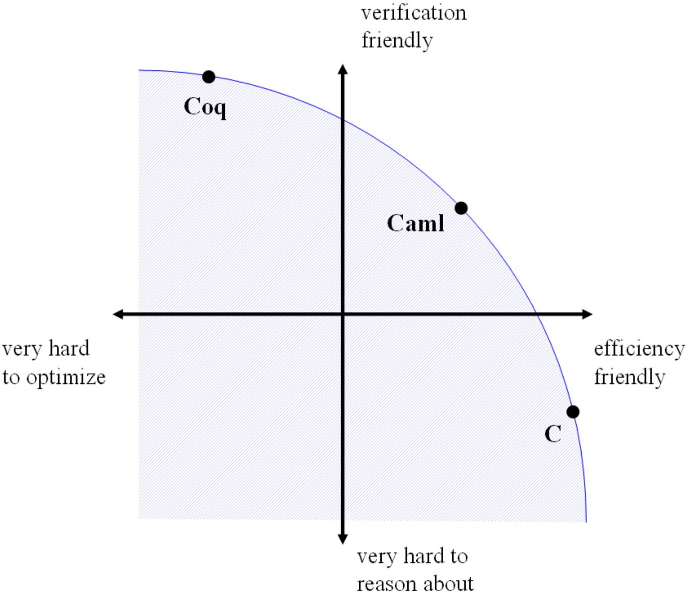
Efficient code:

- support for imperative programming
- compiled with an optimizing compiler

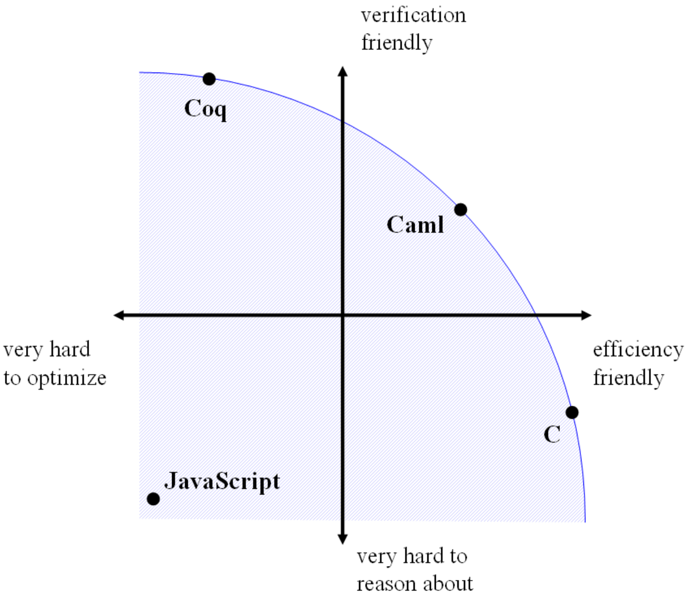
Correctness vs efficiency



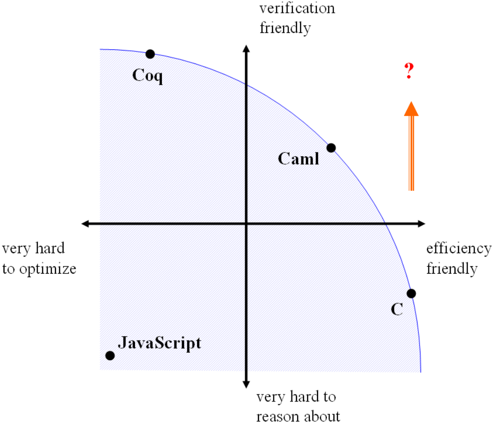
Correctness vs efficiency



Correctness vs efficiency



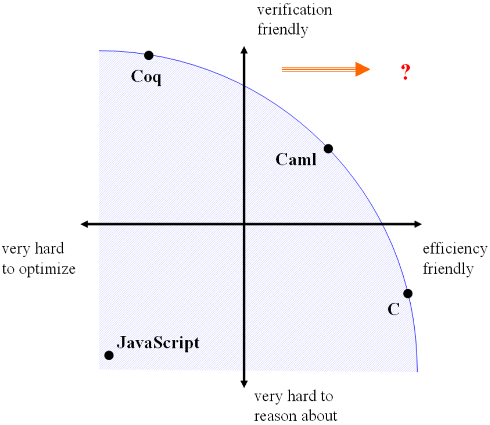
(1) Start from an existing programming language



For a given programming language, what is the best way to reason about programs written in this language?

→ restricting the language and/or developing good reasoning tools

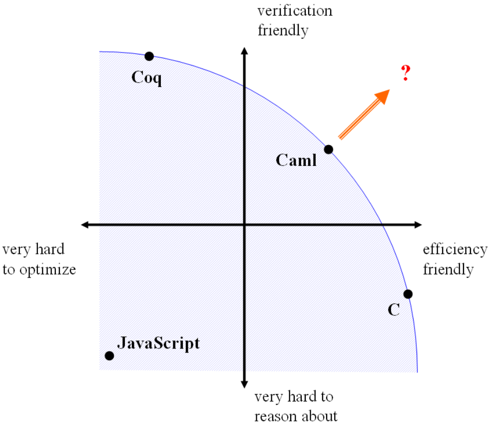
(2) Start from an existing theorem prover



For a given theorem prover, what is the best programming language that can be embedded in this prover?

→ Coq, Agda, Ynot

(3) Design a new programming language



What would be the programming language that allows to describe efficient programs and easily prove them correct?

Verification using CFML

Purely-functional data structures (half of Chris Okasaki's book)

- Batched queue
- Bankers queue
- Physicists queue
- Real-time queue
- Implicit queue
- Bootstrapped queue
- Hood-Melville queue
- Leftist heap
- Pairing heap
- Lazy pairing heap
- Splay heap
- Binominal heap
- Unbalanced set
- Red-black set
- Bottom-up merge sort
- Catenable lists
- Binary random-access lists

Imperative algorithms, data structures and tricky functions

- Dijkstra's shortest path
- Union-Find
- Mutable lists
- Sparse arrays
- List.iter
- compose
- gensym
- CPS-append
- Landin's knot

Verification w.r.t. representation predicates

Specification of insertion in a purely-functional binary tree

$$\forall T. \quad \mathsf{App}\,\mathsf{insert}\,x\,t\,[\mathsf{Btree}\,t\,T]\ (\lambda t'.\,[\mathsf{Btree}\,t'\,(T \cup \{x\})])$$

Specification of insertion in a polymorphic binary tree

$$\forall RTX. \quad \mathsf{App}\,\mathsf{insert}\,x\,t\,[\mathsf{R}\,x\,X\ \wedge\ \mathsf{Btree}\,R\,t\,T]\ (\lambda t'.\,[\mathsf{Btree}\,R\,t'\,(T \cup \{X\})])$$

Removing representation predicates

A much more practical specification for proving programs manipulating sets:

$$\forall T X. \quad \mathsf{App\,insert}\, X\, T\, [\,]\, (\lambda T'.\, [T' = T \cup \{X\}])$$

but we are confusing binary trees with finite sets...

Solution: program directly with mathematical objects (e.g., finite sets) and give hints to tell the compiler which concrete implementation to use

Programming with mathematical objects

Caml implementation (e.g., binary trees)		Coq mathematical model (e.g., finite sets)

before:

source code	$\longrightarrow$ lift using repr. predicates	verification

after:

	$\longleftarrow$ compilation using repr. hints	source code and verification

Summary

- Program using mathematical objects, not concrete implementations:
  - Pure: sequences, sets, maps, graphs, ...
  - Imperative: sequences, sets, maps, graphs, ...
- When needed, indicate the concrete implementation to use
- Prove the correctness of the concrete implementations once and for all
- Enjoy simpler specifications and simpler proofs!

Before: typical Caml code

```
module Pqueue := PriorityQueue(CompareSnd)

let dijkstra g s e =
   let n = Array.length g in
   let b = Array.make n Infinite in
   let v = Array.make n false in
   let q = Pqueue.create() in
   b.(s) <- Finite 0;
   Pqueue.push (s,0) q;
   while not (Pqueue.is_empty q) do
      let (x,dx) = Pqueue.pop q in
      if not v.(x) then begin
         v.(x) <- true;
         let update (y,w) =
            let dy = dx + w in
            if (match b.(y) with | Finite d -> dy < d
                                 | Infinite -> true)
              then (b.(y) <- Finite dy; Pqueue.push (y,dy) q) in
         List.iter update g.(x);
      end;
   done;
   b.(e)
```

After: more abstract Caml code

```
let dijkstra g{adjlist} s e =
   let nodes = Graph.nodes g in
   let b{array} = ImpMap.init_from_set (fun _ -> Infinite) nodes in
   let v{array} = ImpMap.init_from_set (fun _ -> false) nodes in
   let q{priority_queue(compare_snd)} = ImpMultiset.empty () in
   b[s] <- Finite 0;
   ImpMultiset.push (s,0) q;
   while not (ImpMultiset.is_empty q) do
      let (x,dx) = ImpMultiset.pop_min compare_snd q in
      if not v[x] then begin
         v[x] <- true;
         let update (y,w) =
           let dy = dx + w in
           if (match b[y] with | Finite d -> dy < d
                               | Infinite -> true)
             then (b[y] <- Finite dy; ImpMultiset.push (y,dy) q) in
         Set.iter update (Graph.neighbors g x);
      end;
   done;
   B[e]
```