Characteristic Formulae for the Verification of Imperative Programs

Arthur Charguéraud

Max Planck Institute for Software Systems

PPS 2012/03/01

Overview

Proving the correctness of arbitrarily-complex programs

→ given a Caml program

→ given a Coq spec.

→ how can we build a correctness proof?

characteristic formulae

used to describe the semantics of the code in the logic

Interpretation of caracteristic formulae (CF)

$$C \xrightarrow{\hspace{6em}} [\![C]\!]$$

source code without any
modification nor annotation

characteristic formula, expressed in
higher-order logic using ∀,∃,∧,⇒,...

CF are sound and complete w.r.t. Hoare logic

$$\forall H.\forall H'.\quad [\![C]\!]\; H\; H' \quad\Longleftrightarrow\quad \{H\}\; C\; \{H'\}$$

heap predicates
(heap → Prop)

application in
higher-order logic

total correctness
Hoare triple

→ in any heap satisfying *H*, the execution of the code *C* terminates and leaves a heap that satisfies *H'*

Properties

 → **characteristic formulae** → 

→ CF are built automatically

→ CF are built compositionaly

→ CF are of linear size

→ CF support local reasoning (frame rule)

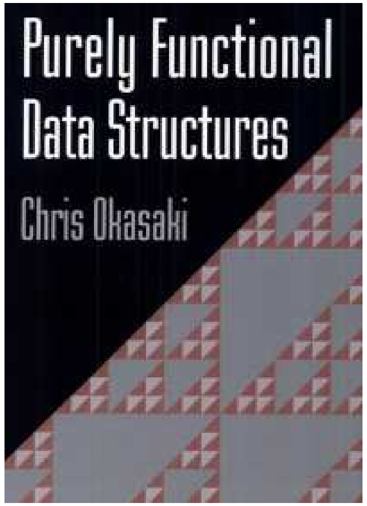
→ CF are displayed in a way that resembles source code

→ CF can be manipulated using solely high-level tactics

Verification using CFML

→ CFML supports core Caml except float, integer modulo, exceptions, objects

Half of 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 higher-order programs

→ Dijkstra's shortest path algorithm, Union-Find, Sparse arrays, Mutable Lists,

→ Local state, e.g., gensym

→ Effectful higher-order functions, e.g., List.iter or compose

→ CPS functions, e.g., CPS-append

→ Functions in the store, e.g. Landin's knot (recursion through the store)

Related work (1/2)

– Not a verification condition generator (VCG)

→ source code needs not be annotated with invariants
→ invariants are instead provided in interactive proofs
→ CF optimized for interactive proofs
(quick fixes to invariants, readable proof obligations)

– Not a shallow embedding

→ Caml functions are not mapped to Coq functions
→ Source language is Caml, not Coq+monad
→ Auxiliary variables not mixed up with code
→ ≠Ynot: less dependend types, no circularity problem

Related work (2/2)

– **Not a dynamic logic** (e.g., Key)

→ no ad-hoc logic construct to embed source code

→ allows to stay in a standard logic and use existing tools

– **Not a deep embedding**

→ no inductive datatype is used to represent code syntax

→ avoids predicate relating Coq values to Caml values

→ avoids issues related to binders and substitutions

Part 2

(1) Introduction: what CF are and what they are not

(2) Example: verification of Dijkstra's algorithm

(3) Technical insight: how to construct CF

Dijkstra's shortest path

```
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)
```

mutable data structures

loop

pattern matching

higher-order function

abstract module

Generated axioms

```
Axiom dijkstra : func.
```

abstract data type used to represent functions

```
Axiom dijkstra_cf :
   (@CFPrint.tag tag_top_fun _ _ (@CFPrint.tag tag_body _ _
   (forall K :  (CFHeaps.loc -> (int -> (int -> ((CFHeaps.hprop
   -> ((_ -> CFHeaps.hprop) -> Prop)) -> Prop)))), ((is_spec_3
   K) -> ((forall g : CFHeaps.loc, (forall s : int, (forall e :
   int, ((((K g) s) e) (@CFPrint.tag tag_let_trm (Label_create
   'n) _ (local (fun H : CFHeaps.hprop => (fun Q : (_ ->
   CFHeaps.hprop) => (Logic.ex (fun Q1 : (int -> CFHeaps.hprop)
   => ((Logic.and (((@CFPrint.tag tag_apply _ _ ((((@app_1
   CFHeaps.loc) int) ml_array_length)...
```

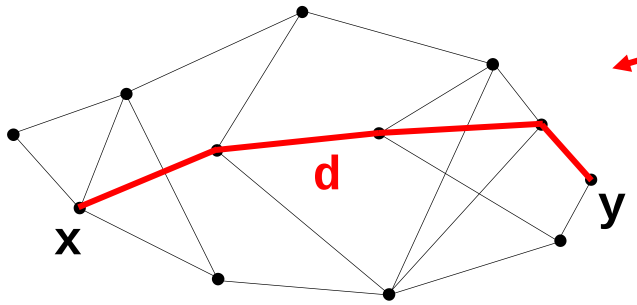
100 more lines like this

```
Print dijkstra_cf.
```

shows something that reads like ML code

→ Axioms justified by the soundness theorem (paper proof)

Specification



mathematical graph *G*

```
Theorem dijkstra_spec : ∀ g x y G,
   nonnegative_edges G ->
   x \in nodes G ->
   y \in nodes G ->
   (App dijkstra g x y)
      (g ~> GraphAdjList G)
      (fun d => [d = dist G x y]
                \* g ~> GraphAdjList G)
```

application

pre-condition

post-condition

(x ~> R) is defined as (R x)

Loop invariant

```
Definition hinv Q B V : hprop :=
   g ~> GraphAdjList G
\* v ~> Array V
\* b ~> Array B
\* q ~> Pqueue Q
\* [inv Q B V].
```

G : graph int
V : array bool
B : array intbar
Q : multiset (int*int)

```
Record inv G s Q B V : Prop := {
 Bdist: ∀x, x \in nodes G -> V\(x) = true ->
          B\(x) = dist G s x;
 Bbest: ∀x, x \in nodes G -> V\(x) = false ->
          B\(x) = mininf weight (crossing V x);
 Qcorr: ∀x, (x,d) \in Q ->
          x \in nodes G /\ ∃p, crossing V x p /\ weight p = d;
 Qcomp: ∀x p, x \in nodes G -> crossing V x p ->
          ∃d, (x,d) \in Q /\ d <= weight p;
 SizeV: length V = n;
 sizeB: length B = n }
```

Main lemma

```
Lemma inv_update : forall L V B Q x y w dx dy,
  x \in nodes G ->
  has_edge G x y w ->
  dy = dx + w ->
  Finite dx = dist G s x ->
  inv (V\(x:=true)) B Q (new_crossing x L V) ->
  If len_gt (B\(y)) dy
    then inv (V\(x:=true)) (B\(y:=Finite dy)) (\{(y, dy)} \u Q) ...
    else inv (V\(x:=true)) B Q (new_crossing x ((y,w)::L) V) .
Proof.
introv Nx Ed Dy Eq [Inv SV SB]. sets_eq V': (V\(x:=true)).
lets NegP: nonneg_edges_to_path Neg.
intros z. lets [Bd Bb Hc Hk]: Inv z. tests (z = y).
(* case z = y *)
forwards~ (px&Px&Wx&Mx): (@mininf_fin
lets Ny: (has_edge_in_nodes_r Ed).
sets p: ((x,y,w)::px).
asserts W: (weight p = dy). subst p.
tests (V'\(y)) as C; case_If as Nlt.
(* subcase y visisted, distance improved *)
false. rewrite~ Bd in Nlt. forwards M: mininf_len_gt Nlt p; subst~ p.
 rewrite weight_cons in M. math.
(* subcase y visisted, distance not improved *)
...
```

no reference to CF in this lemma,
but only mathematical reasoning

all the nontrivial reasoning is here;
180 lines across several lemmas;
8 seconds to type-check

Proof script

```
Theorem dijkstra_spec : ∀ g x y G, ... (App dijkstra g x y) ...
Proof.
xcf. introv Pos Ns De. unfold GraphAdjList at 1.
hdata_simpl. xextract as N Neg Adj. xapp.
intros Ln. rewrite <- Ln in Neg.
xapps. xapps. xapps. xapps*. xapps.
set (data := fun B V Q => g ~> Array N \*
  v ~> Array V \* b ~> Array B \* q ~> Heap Q).
set (hinv := fun VQ => let '(V,Q) := VQ in
 Hexists B, data B V Q \* [inv G n s V B Q (crossing G s V)]).
xseq (# Hexists V, hinv (V,\{})).
set (W := lexico2
           (binary_map (count (= true)) (upto n))
           (binary_map card (downto 0))).
xwhile_inv W hinv. refine (ex_intro' (_,_)).
unfold hinv,data. hsimpl. applys_eq~ inv_start 2.
permut_simpl. intros [V Q]. unfold hinv.
xextract as B Inv. xwhile_body.
unfold data. xapps. xret.
...
Qed.
```

CFML tactics

loop invariant

termination measure

use of a lemma about the invariant

for 20 lines of code, 48 lines of proofs (including 8 lines of invariants); checked in 8 seconds

A typical proof obligation

```
Pos : nonnegative_edges G
Ns : s \in nodes G
Ne : e \in nodes G
Neg : nodes_index G n
Adj : forall x y w,
      x \in nodes G -> Mem (y, w) (N\(x)) = has_edge G x y w
Nx : x \in nodes G
Vx : ~ V\(x)
Dx : Finite dx = dist G s x
Inv : inv G n s V' B Q (new_crossing G s x L' V)
EQ : N\(x) = rev L' ++ (y, w) :: L
Ew : has_edge G x y w
Ny : y \in nodes G
```

hypotheses

```
(Let dy := Ret dx + w in
  Let _x38 := App ml_array_get b y ; in
   If_ Match
       (Case _x38 = Finite d [d] Then Ret (dy '< d) Else
       (Case _x38 = Infinite Then Ret true Else Done))
   Then (App ml_array_set b y (Finite dy) ;) ;;
         App push (y, dy) h ; Else (Ret tt))
(q ~> Pqueue Q \* b ~> Array B \* v ~> Array V' \* g ~> Array N)
(fun _:unit => hinv' L)
```

characteristic formula

pre-condition

post-condition

Part 3

---

(1) Introduction: what CF are and what they are not

(2) Example: verification of Dijkstra's algorithm

(3) Technical insight: how to construct CF

→ characteristic formula for sequences

→ treatment of functions

→ integration of the frame rule

→ relation with denotational semantics

Construction

Hoare logic rule for sequence

$$\frac{\{H\}\ C_1\ \{H''\} \qquad \{H''\}\ C_2\ \{H'\}}{\{H\}\ (C_1\ ;\ C_2)\ \{H'\}}$$

Property of characteristic formulae

$$\forall H.\, \forall H'.\ [\![C_1\ ;\ C_2]\!]\ H\ H' \iff \{H\}\ (C_1\ ;\ C_2)\ \{H'\}$$

Characteristic formula for sequence

$$[\![C_1\ ;\ C_2]\!] \ \equiv\ \lambda H.\, \lambda H'.\ \ \exists H''.\ [\![C_1]\!]\ H\ H'' \ \wedge\ [\![C_2]\!]\ H''\ H'$$

→ from an inductive to a recursive definition of Hoare Logic

Notation

Definition from the previous slide

$$[\![C_1 \,;\, C_2]\!] \;\equiv\; \lambda H.\, \lambda H'.\;\; \exists H''.\; [\![C_1]\!]\, H\, H'' \;\wedge\; [\![C_2]\!]\, H''\, H'$$

Definition of a Coq notation

$$(\mathcal{F}_1 \,;;\, \mathcal{F}_2) \;\equiv\; \lambda H.\, \lambda H'.\;\; \exists H''.\; \mathcal{F}_1\, H\, H'' \;\wedge\; \mathcal{F}_2\, H''\, H'$$

Characteristic formula for sequence, revisited

$$[\![C_1 \,;\, C_2]\!] \;\equiv\; [\![C_1]\!] \,;;\, [\![C_2]\!]$$

→ CF generation is simple, compositional and linear-size

Tactics

View with notation

$$(\mathcal{F}_1 \;;;\; \mathcal{F}_2)\ H\ H'_?$$

a Coq unification variable (evar)

View without notation

$$\exists H''.\ \mathcal{F}_1\ H\ H'' \quad \wedge \quad \mathcal{F}_2\ H''\ H'_?$$

Action of tactic xseq defined as (esplit;split)

$$\mathcal{F}_1\ H\ H''_? \qquad\qquad \mathcal{F}_2\ H''_?\ H'_?$$

After solving first subgoal

$$\mathcal{F}_2\ H''\ H'_?$$

→ CFML can be used without knowledge of CF definitions

Functions

Consider a top-level function definition

$$\mathsf{function}\ f\,(x)\ \{C\}$$

Two axioms are generated (func and App are abstract)

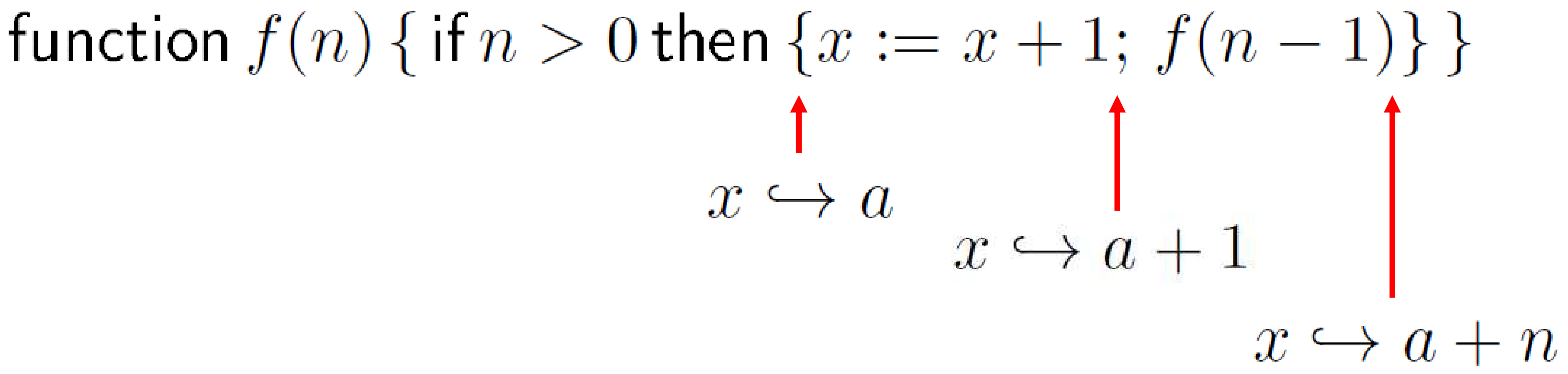
$$\begin{array}{lcl} \mathsf{Axiom}\ f & : & \mathsf{func}. \\ \mathsf{Axiom}\ F & : & \forall x\ H\ H'.\ [\![C]\!]\ H\ H' \ \Rightarrow\ \mathsf{App}\ f\ x\ H\ H'. \end{array}$$

Formula for function calls

$$[\![f\,(v)]\!]\ \equiv\ \lambda H.\,\lambda H'.\ \ \mathsf{App}\ f\ v\ H\ H'$$

Recursive functions

Specification of a recursive functions proved by induction



$$\textsf{function}\ f(n)\ \{\ \textsf{if}\ n > 0\ \textsf{then}\ \{x := x + 1;\ f(n-1)\}\ \}$$

$x \hookrightarrow a$

$x \hookrightarrow a + 1$

$x \hookrightarrow a + n$

Specification

$$\forall n.\ \forall a.\ n \geq 0\ \Rightarrow\ \textsf{App}\ f\ n\ (x \hookrightarrow a)\ (x \hookrightarrow a + n)$$

By induction hypothesis

$$\textsf{App}\ f\ (n-1)\ (x \hookrightarrow a + 1)\ (x \hookrightarrow (a+1) + (n-1))$$

Frame rule

Frame rule is not syntax directed; how to integrate it?

$$\frac{\{H_1\}\ C\ \{H_1'\}}{\{H_1 * H_2\}\ C\ \{H_1' * H_2\}}$$

Insert a predicate at the head of every node in the CF

$$[\![C_1\ ;\ C_2]\!] \;\equiv\; \mathsf{local}\,(\lambda H.\,\lambda H'.\ \ \ldots)$$

$$\mathsf{local}\,\mathcal{F} \;\equiv\; \lambda H.\,\lambda H'.\ \exists H_1 H_1' H_2.\ \begin{cases} \mathcal{F}\, H_1\, H_1' \\ H = H_1 * H_2 \\ H' = H_1' * H_2 \end{cases}$$

→ when no frame is needed, we frame on the empty heap

Types

CF are constructed for code well-typed in ML

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

Algebraic data types are supported

Arbitrary recursive types also, if recursion below an arrow

$$\langle\, \mu A.\ A \rightarrow B \,\rangle \;=\; \mathsf{Func}$$

$$\langle\, \mu A.\ A \times B \,\rangle \;\; \text{not supported}$$

All the rules

Complete set of definitions for ML with side effects

$$
\begin{array}{lcl}
\llbracket v \rrbracket & \equiv & \mathsf{local}\,(\lambda H Q.\ H \rhd Q\,\lceil v \rceil) \\
\llbracket v_1\, v_2 \rrbracket & \equiv & \mathsf{local}\,(\lambda H Q.\ \mathsf{App}\,\lceil v_1 \rceil\,\lceil v_2 \rceil\, H\, Q) \\
\llbracket \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \rrbracket & \equiv & \mathsf{local}\,(\lambda H Q.\ \exists Q'.\ \llbracket t_1 \rrbracket\, H\, Q' \wedge\ \forall x.\ \llbracket t_2 \rrbracket\,(Q'\,x)\,Q) \\
\llbracket t_1\,;\,t_2 \rrbracket & \equiv & \mathsf{local}\,(\lambda H Q.\ \exists Q'.\ \llbracket t_1 \rrbracket\, H\, Q' \wedge \llbracket t_2 \rrbracket\,(Q'\,tt)\,Q) \\
\llbracket \mathsf{let\ rec}\ f = \Lambda\overline{A}.\lambda x.t_1\ \mathsf{in}\ t_2 \rrbracket & \equiv & \mathsf{local}\,(\lambda H Q.\ \forall f.\ (\forall \overline{A} x H' Q'.\ \llbracket t_1 \rrbracket\, H'\, Q' \Rightarrow \mathsf{App}\,f\,x\,H'\,Q') \Rightarrow \llbracket t_2 \rrbracket\, H\, Q) \\
\llbracket \mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \rrbracket & \equiv & \mathsf{local}\,(\lambda H Q.\ (\lceil v \rceil = \mathsf{true} \Rightarrow \llbracket t_1 \rrbracket\, H\, Q) \wedge (\lceil v \rceil = \mathsf{false} \Rightarrow \llbracket t_2 \rrbracket\, H\, Q)) \\
\llbracket \mathsf{crash} \rrbracket & \equiv & \mathsf{local}\,(\lambda H Q.\ \mathsf{False}) \\
\llbracket \mathsf{let}\ x = \Lambda\overline{A}.\,v\ \mathsf{in}\ t \rrbracket & \equiv & \mathsf{local}\,(\lambda H Q.\ \forall x.\ x = \lambda\overline{A}.\lceil v \rceil \Rightarrow \llbracket t \rrbracket\, H\, Q)
\end{array}
$$

For each construct: formula + notation + tactic

CF for purely functional code

→ No pre-condition needed for total correctness

→ CF describes the set of valid post-conditions

$$\llbracket t \rrbracket \;:\; (T \to \mathsf{Prop}) \to \mathsf{Prop}$$

$$\begin{array}{lcl}
\llbracket v \rrbracket & \equiv & \lambda P.\; P\, v \\
\llbracket v\, v' \rrbracket & \equiv & \lambda P.\; \mathsf{App}\, v\, v'\, P \\
\llbracket \mathsf{let}\; x = t\; \mathsf{in}\; t' \rrbracket & \equiv & \lambda P.\; \exists P'.\, \llbracket t \rrbracket\, P' \;\wedge\; (\forall x.\; P'\, x \Rightarrow \llbracket t' \rrbracket\, P) \\
\llbracket \mathsf{let}\; f = \lambda x.\, t\; \mathsf{in}\; t' \rrbracket & \equiv & \lambda P.\; \forall f.\; (\forall x P'.\; \llbracket t \; \rrbracket\, P' \Rightarrow \mathsf{App}\, f\, x\, P') \Rightarrow \llbracket t' \rrbracket\, P
\end{array}$$

In fact, contexts and translations to go from Caml to Coq

$$\llbracket v \rrbracket^{\Gamma} \;\equiv\; \lambda P.\; P\, \lceil v \rceil^{\Gamma}$$

$$\llbracket \mathsf{let}\; x = t\; \mathsf{in}\; t \rrbracket^{\Gamma} \equiv \lambda P.\; \exists P'.\, \llbracket t \rrbracket^{\Gamma}\, P' \;\wedge\; (\forall X.\, P'\, X \Rightarrow \llbracket t' \rrbracket^{(\Gamma, x \mapsto X)}\, P)$$

CF and denotational semantics

Re-interpreting post-conditions as set of objects

$$[\![t]\!]\,P \quad\Leftrightarrow\quad \mathcal{D}(t) \in P$$

Re-interpreting the definition of CF

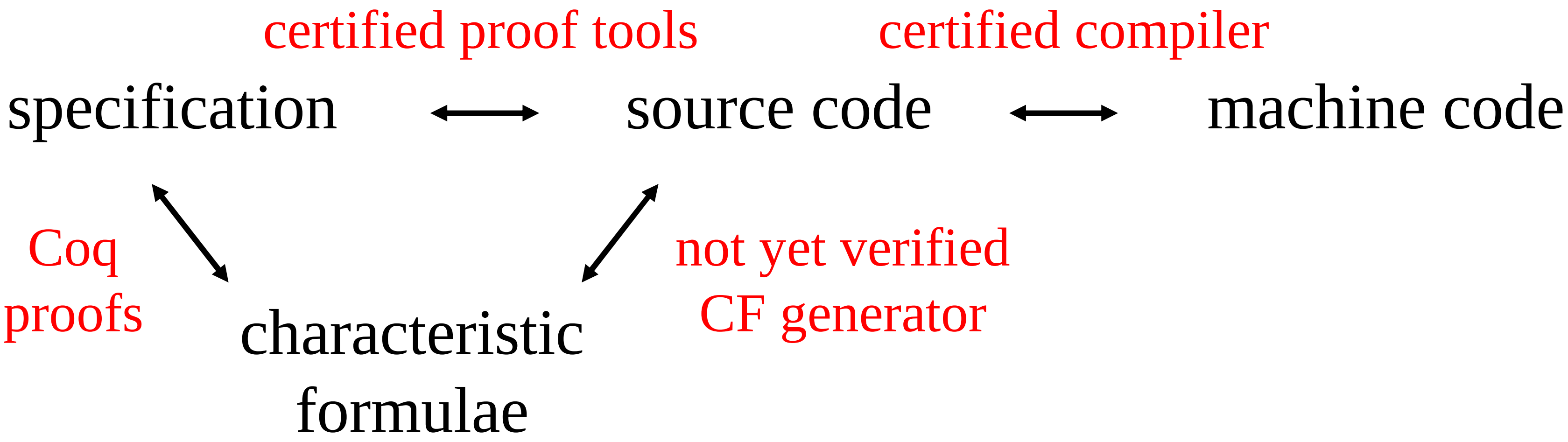
$$\mathsf{App}\,V\,V'\,P \quad\equiv\quad V(V') \in P$$

$$\begin{array}{ll}
\mathcal{D}^{\Gamma}(v) \in P & \Leftrightarrow \mathcal{D}^{\Gamma}(v) \in P \\
\mathcal{D}^{\Gamma}(v\,v') \in P & \Leftrightarrow \big(\mathcal{D}^{\Gamma}(v)\big)\,\big(\mathcal{D}^{\Gamma}(v')\big) \in P \\
\mathcal{D}^{\Gamma}(\mathsf{let}\,x = t\,\mathsf{in}\,t') \in P & \Leftrightarrow \exists P'.\,\mathcal{D}^{\Gamma}(t) \in P' \,\wedge\, (\forall X \in P'.\,\mathcal{D}^{(\Gamma,x\mapsto X)}(t') \in P) \\
\mathcal{D}^{\Gamma}(\mathsf{let}\,f = \lambda x.\,t\,\mathsf{in}\,t') \in P & \Leftrightarrow \forall F.\,(\forall x P'.\,\mathcal{D}^{(\Gamma,x\mapsto X)}(t) \in P' \Rightarrow F(X) \in P') \Rightarrow \mathcal{D}^{(\Gamma,f\mapsto F)}(t') \in P
\end{array}$$

Logically equivalent to

$$\begin{array}{ll}
\mathcal{D}^{\Gamma}(\mathsf{let}\,x = t\,\mathsf{in}\,t') & = \mathcal{D}^{(\Gamma,x\mapsto\mathcal{D}^{\Gamma}(t))}(t') \\
\mathcal{D}^{\Gamma}(\mathsf{let}\,f = \lambda x.\,t\,\mathsf{in}\,t') & = \{\mathcal{D}^{(\Gamma,f\mapsto F)}(t') \,|\, \forall X.\,F(X) = \mathcal{D}^{(\Gamma,x\mapsto X)}(t)\}
\end{array}$$

Verification of the CF generator



certified proof tools

certified compiler

specification ⟷ source code ⟷ machine code

Coq proofs

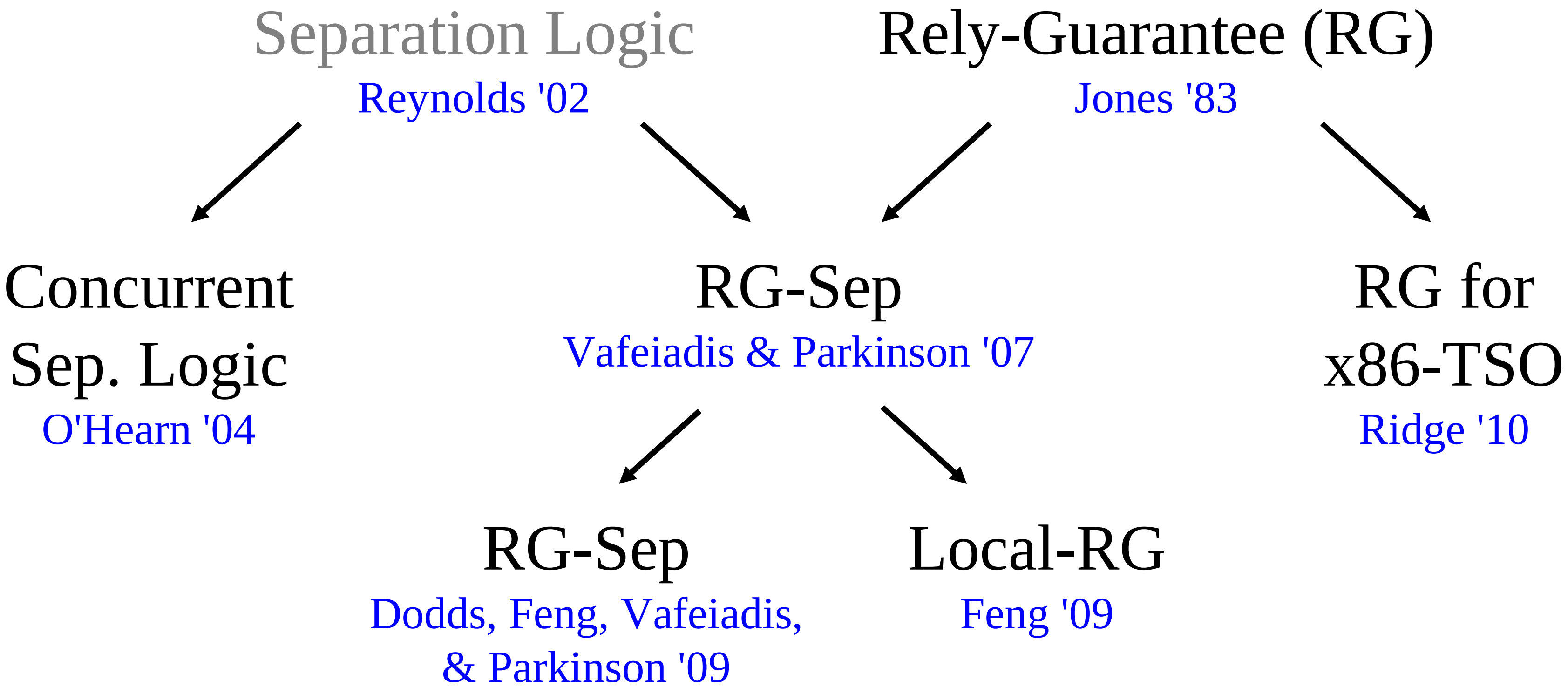
not yet verified CF generator

characteristic formulae

→ CF generator as a Coq function, for a toy language

→ but need a deep embedding of Coq to reason about inductive defs, polymorphism (∀A:Type), modules

Concurrent program logics



Separation Logic
Reynolds '02

Rely-Guarantee (RG)
Jones '83

Concurrent
Sep. Logic
O'Hearn '04

RG-Sep
Vafeiadis & Parkinson '07

RG for
x86-TSO
Ridge '10

RG-Sep
Dodds, Feng, Vafeiadis,
& Parkinson '09

Local-RG
Feng '09

→ good progress, yet still limited and not implemented

Extension to concurrency

I would like to extend CF to support:

→ modular and local reasoning for private resources, shared resources, and also content of write buffers

→ transitions from private to shared and back

→ verification of sequential terms with minimal overhead

→ simple high-level reasoning rules for, e.g., fork-join

Thanks!

Further information

→ ICFP'10 and ICFP'11 papers, my thesis for the proofs

→ examples available on my webpage

→ download and try CFML (open source)

CF and CPS

Term	CPS	CF with App = id
v	$\lambda k.\ k\, v$	$\lambda k.\ k\, v$
$v\, v'$	$\lambda k.\ v\, v'\, k$	$\lambda k.\ v\, v'\, k$
$\mathsf{let}\ x = t\ \mathsf{in}\ t'$	$\lambda k.\ \mathsf{let}\ \boxed{k' = \lambda x.\ \llbracket t' \rrbracket\, k}\ \mathsf{in}\ \llbracket t \rrbracket\, k'$	$\lambda k.\ \boxed{\exists k'}.\ \llbracket t \rrbracket\, k' \ \wedge\ \boxed{(\forall x.\ k'\, x \Rightarrow \llbracket t' \rrbracket\, k)}$
$\mathsf{let}\ f = \lambda x.\, t\ \mathsf{in}\ t'$	$\lambda k.\ \mathsf{let}\ \boxed{f = \lambda x.\ \llbracket t \rrbracket}\ \mathsf{in}\ \llbracket t' \rrbracket\, k$	$\lambda k.\ \forall f.\ \boxed{(\forall x k'.\ \llbracket t' \rrbracket\, k' \Rightarrow f\, x\, k')} \Rightarrow \llbracket t' \rrbracket\, k$

weakening

$$\boxed{\forall k'.\ k'\, x = \llbracket t' \rrbracket\, k}$$

$$\boxed{\forall x k'.\ f\, x\, k' = \llbracket t \rrbracket\, k'}$$

→ CF use implication instead of equality, for weakening

→ Weakening is crucial for abstract data types