FStar: A Higher-Order Effectful Language Designed for Program Verification ?

Shenghao YUAN

University of Rennes1

shenghao.yuan@inria.fr

December 12, 2020

Overview

Program verification: Shall the twain ever meet?

Interactive proof assistants			Semi-automated verifiers of imperative programs	
Coq,	CompCert,	*air*	Dafny,	Verve,
Isabelle,	4 colors,		FramaC,	IronClad,
Agda,	seL4,		Why3	miTLS
Lean,		*gap*		Vale

- **In the left corner**: Very expressive logics (higher-order and often dependently-typed), but purely functional
- **In the right**: effectful programming, SMT-based automation, but only first-order logic

source:Verified Effectful Programming in F

Bridging the gap: F*

- **Functional programming language with effects**
  - like OCaml, F#, Haskell, ...
  - F* extracted to OCaml or F# by default
  - subset of F* compiled to efficient C code
- **Program verifier based on WPs and SMT**
  - like Dafny, FramaC, Why3, ...
- **Interactive proof assistant based on dependent types**
  - like Coq, Lean, Agda, ...
- Other tools in this space:
  - DML/ATS, HTT, Idris, Trellys/Zombie, CoqHammer, PML_2, ...

$$\text{F*} \approx \text{ML} + \text{Coq(subset)} + \text{SMT(z3)}$$

source:Verified Effectful Programming in F

The functional core of F*

- ML-style programs
- refinement type
- dependent function types
- inductive types and Proving termination

```
val fibonacci : nat -> Tot nat
let rec fibonacci n = if n <= 1 then 1
  else fibonacci (n - 1) + fibonacci (n - 2)

val gta : n:nat{n >= 2} -> Lemma (fibonacci n >= n)
let rec gta n =
  match n with
  | 2 -> ()
  | _ -> gta (n-1)
```

ML-style programs

- let-in

```
let f x =
 let y = x*3 in
  x + y
```

- if-then-else

```
let f x = if x > 3 then x else x + 1
```

- let rec

```
let rec fibonacci n = let b = (n <= 1) in
  match b with
  | true -> 1
  | false -> fibonacci (n - 1) + fibonacci (n - 2)
```

Refinement type

refinement type

a refinement type in F* has the form `x:t{phi(x)}`, a refinement of the type `t` to those elements `x` that satisfy the formula `phi(x)`

```
val incr : x:int -> y:int{y = x + 1}
```

Refinement type

refinement type

a refinement type in F* has the form `x:t{phi(x)}`, a refinement of the type `t` to those elements `x` that satisfy the formula `phi(x)`

```
val incr : x:int -> y:int{y = x + 1}
```

Coq sigma-types: `(sig A P)`, or more suggestively `{x:A | P x}`, denotes the subset of elements of the type `A` which satisfy the predicate `P`.

Refinement type

refinement type

a refinement type in F* has the form `x:t{phi(x)}`, a refinement of the type `t` to those elements `x` that satisfy the formula `phi(x)`

```
val incr : x:int -> y:int{y = x + 1}
```

Coq sigma-types: `(sig A P)`, or more suggestively `{x:A | P x}`, denotes the subset of elements of the type `A` which satisfy the predicate `P`.

- OCaml:

```
# List.hd;;
val List.hd (-) : 'a list -> 'a = <fun>
# List.hd [];;
Exception: Failure "hd".
```

- Fstar:

```
val hd : l : list 'a{length l > 0} -> 'a
```

Dependent function types

Dependent function types

a dependent function type in F* is that the type of the result depends on the value of its parameters.

```
val incr : x:int -> y:int{y = x + 1}
let incr x = x + 1
```

General form of function types in F*

x1:t1 → … → En xn:tn[x1 … xn-1] → E t[x1 … xn]

where, (e.g. `val incr :  x:int → Tot y:int{y = x + 1}`)

- x_i: variables (i.e. functions)
- t_i: types (e.g. int, nat ...)
- E_n: effects (e.g. Tot, Dv, St)
- $t[x_1 ... x_m]$ indicates that the variables $x_1 ... x_m$ may appear free in t.

Inductive types and Proving termination

```
type list 'a =
  | Nil : list 'a
  | Cons : hd:'a -> tl:list 'a -> list 'a

val length: list 'a -> Tot nat
let rec length l = match l with
  | [] -> 0
  | _ :: tl -> 1 + length tl
```

e.g. `let mylist = Cons 1 (Cons 2 (Cons 3 Nil))` (i.e. '[1; 2; 3]')
`length mylist` returns 3

Termination:based on well-founded ordering on expressions (<<)

- naturals related by < (negative integers unrelated)
- inductives related by subterm ordering (v = D v1 ... vn => vi << v)
- lex tuples %[a;b;c] with lexicographic ordering

Monadic effects in F*

effect: weakest precondition calculas + monad

programs may have side-effects, such as non-termination, state and exceptions.

- Tot, the effect of a computation that terminates and evaluates to a t-typed result (Tot t)
- Dv, the effect of a computation that may diverge;
- ST, the effect of a computation that may diverge, read, write or allocate new references in the heap;

Monadic effects in F*

effect: weakest precondition calculas + monad

programs may have side-effects, such as non-termination, state and exceptions.

- Tot, the effect of a computation that terminates and evaluates to a t-typed result (Tot t)
- Dv, the effect of a computation that may diverge;
- ST, the effect of a computation that may diverge, read, write or allocate new references in the heap;
- Value types (t): `int`, `list int`, ...
- Computation types (C): `Tot t` and `Dv t`

Monadic effects in F*

effect: weakest precondition calculas + monad

programs may have side-effects, such as non-termination, state and exceptions.

- Tot, the effect of a computation that terminates and evaluates to a t-typed result (Tot t)
- Dv, the effect of a computation that may diverge;
- ST, the effect of a computation that may diverge, read, write or allocate new references in the heap;
- Value types (t): `int`, `list int`, ...
- Computation types (C): `Tot t` and `Dv t`
- Dependent function types: `x:t` $\rightarrow$ `C`

Monadic effects in F*

effect: weakest precondition calculas + monad

programs may have side-effects, such as non-termination, state and exceptions.

- Tot, the effect of a computation that terminates and evaluates to a t-typed result (Tot t)
- Dv, the effect of a computation that may diverge;
- ST, the effect of a computation that may diverge, read, write or allocate new references in the heap;

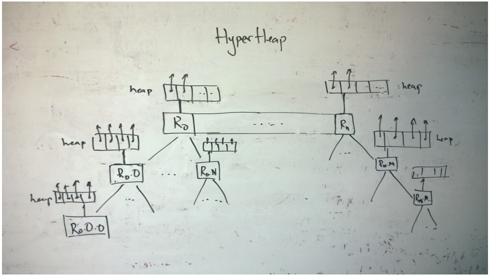
- Value types (t): `int`, `list int`, ...
- Computation types (C): `Tot t` and `Dv t`
- Dependent function types: `x:t` $\rightarrow$ `C`
- Refined value types: `x:t{p}`
- Refined computation types: `Pure t pre post` and `Div t pre post`

LowStar: FStar + C

- C memory model
- stack- and heap-allocated arrays (e.g. LowStar.Buffer)
- a few system-level functions from the C standard library

HyperHeap

A hyper-heap provides organizes the heap into many disjoint fragments, or regions. Each region is collectively addressed by a region identifier, and these identifiers are organized in a tree-shaped hierarchy.



LowStar: FStar + C

HyperStack

Low* refines the HyperHeap memory model, adding a distinguished set of regions that model the C call stack.

```
let main (): ST unit (requires fun h0 -> True)
                     (ensures fun h0 _ h1 -> True) =
  push_frame();
  let images:B.buffer rb_hdr_t = B.alloca ... in
  let slot = choose_image images in
  pop_frame();
  begin match slot with
        | Some t -> ()
        | None -> P.(printf "no suitable image found
           \n" done)
  end
```

Case Study: RIOT-bootloader in LowStar (requirement)

riotboot

riotboot expects the flash to be formatted in slots

1. Choosing the image with the latest version
   1. validating the image header using the fletcher32 algorithm
   2. if *is_valid*, choosing the latest one.
2. Booting into the image in slot

Header

The header contains "RIOT" as a magic number to recognize a RIOT firmware image, a checksum, an start address, and the version of the RIOT firmware

```
type rb_hdr_t = { magic_number : UInt32.t;
                  version : UInt32.t;
                  start_addr : UInt32.t;
                  chksum : UInt32.t}
```

Case Study: RIOT-bootloader in LowStar (Main)

```
let main () : ST unit (requires fun _ -> True)
                      (ensures fun h _ h' -> True) =
  push_frame();
  let images:B.buffer rb_hdr_t = B.alloca ... in
  let slot = choose_image images in (*choose_image*)
  pop_frame();
  begin match slot with
      | Some t -> () (*jump_to_image*)...
```

```
void cpu_jump_to_image(uint32_t image_address){
  __set_MSP(*(uint32_t*)image_address);
  image_address += 4;
  uint32_t destination_address = *(uint32_t*)
     image_address;
  destination_address |= 0x1;
  __asm("BX %0" :: "r" (destination_address));
}
```

choose_image in LowStar

- pre-condition: the liveness of images is True
- post-condition: the liveness of images is True

```
val choose_image : images:B.buffer rb_hdr_t{B.length
    images == rb_slot_numof} -> ST (option UInt32.t)
  (requires (fun h0 -> B.live h0 images))
  (ensures (fun h0 _ h1 -> B.live h1 images))

let choose_image images = choose_image_aux images (
   rb_slot_numof - 1) None

val choose_image_aux : images:B.buffer rb_hdr_t ->
   len:int{len < B.length images /\ len >= 0} ->
   option rb_hdr_t -> ST (option UInt32.t)
  (requires (fun h0 -> B.live h0 images))
  (ensures (fun h0 _ h1 -> B.live h1 images))
  (decreases len)
```

choose_image in LowStar

```
let rec choose_image_aux images len opt =
  match len with
  | 0 -> begin match opt with
         | Some t -> Some (t.start_addr)
         | None -> None
         end
  | _ -> let img = images.(UInt32.uint_to_t len) in
let b = choose_image_aux1 img images in (*here!*)
  if b = true then
   match opt with
   | None -> choose_image_aux images (len -1) (Some
      img)
   | Some t -> if img.version <= t.version then
      choose_image_aux images (len -1) opt else
      choose_image_aux images (len - 1) (Some img)
  else choose_image_aux images (len - 1) opt
```

choose_image in LowStar

- pre-condition: the liveness of images is True
- post-condition: the liveness of images is True and images modifies nothing!

```
val choose_image_aux1 : rb_hdr_t -> images:B.buffer
    rb_hdr_t -> ST bool
  (requires (fun h0 -> B.live h0 images))
  (ensures (fun h0 _ h1 -> B.modifies B.loc_none h0
      h1 /\ B.live h1 images))

let choose_image_aux1 img images =
  push_frame ();
  let tb = B.alloca img 1ul in
  let b:bool = (Lowhdr.rb_hdr_validate tb = 0) in (*
      is_valid()*)
  pop_frame ();
  b
```

slot/hdr in LowStar

- pre-condition: the liveness of the pointer *h* is True
- post-condition: the liveness of images is True and *h* modifies nothing!

```
val rb_hdr_validate : h:B.buffer rb_hdr_t{B.length h
    == 1} -> ST int
  (requires (fun h0 -> B.live h0 h))
  (ensures (fun h0 _ h1 -> B.live h1 h /\ B.modifies
      B.loc_none h0 h1))

let rb_hdr_validate h =
  let h1 = h.(0ul) in
  let hc = rb_hdr_checksum h in (*checksum!*)
  if (h1.magic_number = rm) && (hc = h1.chksum) then
   0
  else
   -1
```

slot/hdr in LowStar

- pre-condition: the liveness of the pointer *h* is True
- post-condition: the liveness of images is True and *h* modifies nothing!

```
val rb_hdr_checksum_aux : rb_hdr_t -> ST UInt32.t
(fun h0 -> True) (fun h0 _ h1 -> B.modifies B.
   loc_none h0 h1)
let rb_hdr_checksum_aux h =
  push_frame ();
  let tb = B.alloca 0us 8ul in
  rb_hdr_t2uint16_t h tb;
  let res = LowFletcher32.fletch32 offset_chksum tb
     in (*fletch32 algorithm*)
  pop_frame ();
  res
  ...
let rb_hdr_checksum b= rb_hdr_checksum_aux b.(0ul)
```

fletcher32 in LowStar

- pre-condition: the liveness of the pointer *d* is True
- post-condition: the liveness of images is True and *d* modifies nothing!

```
val fletch32 : words:UInt16.t{words>=0us} -> d:B.
    buffer UInt16.t{B.length d > UInt16.v words} ->
    ST UInt32.t
  (requires (fun h0 -> B.live h0 d))
  (ensures (fun h0 _ h1 -> B.live h1 d /\ B.modifies
      B.loc_none h0 h1))
let fletch32 words d =
  let (sum1,sum2) = while_t words d 0xfffful 0
     xfffful in
  let sum11 = UInt32.add_mod (sum1 &^ 0xfffful) (
     sum1 >>^ 16ul) in
  let sum21 = UInt32.add_mod (sum2 &^ 0xfffful) (
     sum2 >>^ 16ul) in
  (sum21 <<^ 16ul) |^ sum11
```

Troubles

Thank You Very Much!

Merci Beaucoup!

Official Sponsors