Protocols course

*Low-level and stateful programming using F**

Jonathan Protzenko (MSR)
jonathan.protzenko@gmail.com

Points covered in this lecture

- Pure programming, a word about Z3, an example
- Stateful programming, why it's painful
- A structured memory model (hyperheaps, framing)
- A low-level memory model (hyperstacks, liveness)
- Extraction to C, an example

Please ask questions!

Pure programming

Pure computations

```
let abs x = if x >= 0 then x else -x
```

- This computation is pure
- This computation is total

Pure computations

```
let abs x = if x >= 0 then x else -x
```

- This computation is pure
- This computation is total

We should mark it as such.

Total computations can be used in specifications

```
let abs x: Tot int =
  if x >= 0 then x else -x

let abs' x: Tot (y:int{ y = abs x }) =
  if x > 0 then x else -x
```

- Notice the refinement type above
- Total functions allow the programmer to reason about values, not computations

A reminder: intrinsic *vs.* extrinsic

```
(* Intrinsic *)
let abs' x: Tot (y:int{ y = abs x }) =
  if x > 0 then x else -x

(* Extrinsic *)
let lemma_abs_abs' ():
  Lemma (forall (x: int). abs x = abs' x) =
  ()
```

- Intrinsic: refinements or pre/post-conditions
- Extrinsic: lemmas that reveal properties after the definition

Total code can be evaluated

F* can evaluate total computations at type-checking time:

```
module L = FStar.List.Tot

let _ =
  let l = [ -1; -2 ] in
  let l' = L.map abs l in
  assert_norm (L.hd l' = 1);
  assert_norm (L.hd (L.tl l') = 2);
  ()

let _ =
  assert_norm (pow2 20 = 1048576)
```

This is the normalizer; life is easy when using total computations.

Programming in pure style

```
let _ =
  let l = abs (-1) in
  ...
  (* plenty of code *)
  ...
  l
```

Immutability is good for reasoning

Programming in pure style (2)

```
let _ =
  let l = [] in
  let l = 1 :: l in
  let l = L.tl l in
  assert (l = [])
```

Shadowing alleviates the need to reason about mutable state

An exercise in pure style

One-time pad encryption and decryption, a core operation in cryptography.

$$\begin{aligned} x \oplus x &= 0 \\ x \oplus y &= y \oplus x \\ x \oplus (y \oplus z) &= (x \oplus y) \oplus z \end{aligned}$$

Binary or (a.k.a. `xor`) is a commutative, associative operation.

An exercise in pure style

One-time pad encryption and decryption, a core operation in cryptography.

$$
\begin{aligned}
x \oplus x &= 0 \\
x \oplus y &= y \oplus x \\
x \oplus (y \oplus z) &= (x \oplus y) \oplus z
\end{aligned}
$$

Binary or (a.k.a. `xor`) is a commutative, associative operation.

A very enjoyable property: $\boxed{w \oplus (w \oplus p) = p}$

An exercise in pure style (2)

One-time pad encryption and decryption, a core operation in cryptography.

$$\begin{aligned}
&\text{encrypt}(w^*, p^*) = w_0 \oplus p_0 \dots w_n \oplus p_n \\
&\text{decrypt}(w^*, c^*) = w_0 \oplus c_0 \dots w_n \oplus c_n \\
&\text{decrypt}(w^*, \text{encrypt}(w^*, p^*)) = \\
&\qquad w_0 \oplus (w_0 \oplus p_0) \dots w_n \oplus (w_n \oplus p_n) = \\
&\qquad p_0 \dots p_n = p^*
\end{aligned}$$

Encrypt and decrypt are the same; the one-time pad w^* is usually derived from a key (possibly derived with asymmetric cryptography) and a nonce.

An exercise in pure style (2)

Goal: implement these three functions.

```
module U32 = FStar.UInt32

val encrypt: otp:list U32.t -> plain:list U32.t ->
  Tot (list U32.t)
val decrypt: otp:list U32.t -> plain:list U32.t ->
  Tot (list U32.t)
val encrypt_decrypt: otp:list U32.t -> plain:list U32.t ->
  Lemma (decrypt otp (encrypt otp plain) = plain)
```

Skeleton at `http://jonathan.protzenko.fr/mpri/`

Syntax: `U32.( x ^^ y )` for XOR

An exercise in pure style (3)

Recap:

- F* can reason efficiently about pure functions;
- these are encoded into the SMT solver (Z3);
- the definitions can be unfolded by the SMT-solver

Pure *vs.* Tot

```
module U32 = FStar.UInt32
module L = FStar.List.Tot

let rec encrypt
  (otp: list U32.t)
  (plain: list U32.t{ L.length plain = L.length otp }):
  Tot (cipher: list U32.t{ L.length cipher = L.length plain })

(* versus *)

val encrypt: otp:list U32.t -> plain:list U32.t ->
  Pure (list U32.t)
    (requires (L.length otp = L.length plain))
    (ensures (fun cipher -> L.length otp = L.length cipher))
```

Tot is Pure with trivial (e.g. True) pre- and post-conditions.

Understanding the difference between pure and effectful

Pure and effectful have a wildly different status in F*.

`fstar —log_queries test.fst`

```
let abs x: Tot nat = if x >= 0 then x else -x
```

```
;;;;;;;;;;;;;;;Equation for Test.abs
(assert (!
(forall ((@x0 Term))
 (! (= (Test.abs @x0)
(ite (= (Prims.op_GreaterThanOrEqual @x0
(BoxInt 0))
(BoxBool true))
@x0
(ite true
(Prims.op_Minus @x0)
Tm_unit)))
```

Understanding the difference between pure and effectful (2)

- Pure functions are encoded to the SMT solver
- The SMT solver can reason about them and reduce them
- No such thing happens for effectful computations

Effectful programming with `Heap`

Understanding the difference between pure and effectful (3)

```
let abs x: Tot nat = if x >= 0 then x else -x

let abs_ref x: St nat =
  let r = ST.alloc x in
  if !r < 0 then
    r := - !r;
  !r

(* Works: SMT-solver "knows what abs is" *)
let test =
  assert (abs (-1) = 1)

(* Fails SMT-solver "doesn't know what abs_ref is" *)
let test' =
  assert (abs_ref (-1) = 1)
```

Understanding the difference between pure and effectful (4)

```
let abs x: Tot nat = if x >= 0 then x else -x

(* Function has a Hoare Triple *)
let abs_ref' x: ST int
  (requires (fun _ -> True))
  (ensures (fun _ y _ -> y = abs x))
=
  let r = ST.alloc x in
  if !r < 0 then
    r := - !r;
  !r

(* Works: SMT-solver reasons about the logical specification *)
let test' =
  assert (abs_ref' (-1) = 1)
```

Aside: how does an SMT-solver work?

F* constructs a logical formula ϕ that has to be valid.

- That is, $\forall$ variables, ϕ is always true (validity).
- That is, $\forall$ variables, $\neg\phi$ is always false.

SMT-solvers are about satisfiability, that is:

- There exists an assignment of variables that satisfies this formula (SAT), or:
- There exists no assignment of variables that satisfies this formula (UNSAT).

If $\neg\phi$ is UNSAT, then $\neg\phi$ is always false, i.e. the program is correct.

Pure *vs.* stateful

```
(* Short-form effect *)
let abs1 x: Tot nat =
  ...

(* Long-form effect *)
let abs2 x: Pure int
  (requires True)
  (ensures (fun y ->
    y >= 0)) =
  ...
```

```
(* Short-form effect *)
let abs3 x: St nat =
  ...

(* Long-form effect *)
let abs4 x: ST nat
  (requires (fun _ -> True))
  (ensures (fun _ y _ ->
    y = abs x)) =
  ...
```

Note the difference in pre- and post-conditions.

Why is stateful verification hard?

```
module U32 = FStar.UInt32

type connection = {
  nonce: ref U32.t;
  count: ref U32.t;
  ...
}

let nonce c: St U32.t =
  !c.nonce

let test =
  let c1 = { nonce = alloc 0ul; key = alloc []; count = alloc 0ul
  let n = nonce c1 in
  let h0 = ST.get () in
  (* Fails! Why? *)
  assert (Heap.sel h0 c1.count = 0ul)
```

Why is stateful verification hard? (2)

The heap is modeled at the verification level:

```
(* FStar.Heap.fst *)
assume new type heap : Type0

assume val sel: #a:Type -> heap -> ref a -> GTot a
assume val upd: #a:Type -> heap -> ref a -> a -> GTot heap
assume val emp: heap
```

- The effect system of F* has a model of the heap at each program point (monadic transformation)
- GTot means “for proofs only”
- Unlike pure values, references may change: we no longer know what the reference is!
- The model is written by experts – don’t want to assume false
- Ideally, we have an implementation of the model (map)

Why is stateful verification hard? (3)

The heap is modeled at the verification level:

```
(* FStar.Heap.fst *)

type modifies (mods:set aref) (h:heap) (h':heap) =
  equal h' (concat h' (restrict h (complement mods))) /\
  subset (domain h) (domain h')

abstract val lemma_modifies_trans:
  m1:heap -> m2:heap -> m3:heap ->
  s1:set aref -> s2:set aref ->
  Lemma
    (requires (modifies s1 m1 m2 /\ modifies s2 m2 m3))
    (ensures (modifies (union s1 s2) m1 m3))
```

Why is stateful verification hard? (4)

The heap is modeled at the verification level:

- References are always live
- No need to have liveness predicates
- This is the garbage-collected, ML-style heap.
- No free operation, no notion of lifetime.

Ideally, we would prove a garbage-collector to show that this is a sound model.

These assumptions will change later on in the lecture.

Why is stateful verification hard? (5)

Need to talk about modifies clauses.

```
let nonce c: ST U32.t
  (requires (fun _ -> True))
  (ensures (fun h_0 _ h_1 ->
    modifies !{} h_0 h_1)) =
  !c.nonce
```

Where `!{ r1; r2 ... }` is syntax for a set of references.

Why is stateful verification hard? (6)

This does not verify. Why?

```
let bump (c_in: connection) (c_out: connection): ST unit
  (requires (fun _ -> True))
  (ensures (fun h_0 _ h_1 ->
    let open U32 in
    Heap.sel h_1 c_in.count =^ Heap.sel h_0 c_in.count +%^ 1ul /\
    Heap.sel h_1 c_out.count =^ Heap.sel h_0 c_out.count +%^ 1ul)
  let open U32 in
  c_in.count := !c_in.count +%^ 1ul;
  c_out.count := !c_out.count +%^ 1ul
```

Why is stateful verification hard? (7)

Not only the **`modifies`** clauses, but also the **`disjoint`** clauses.

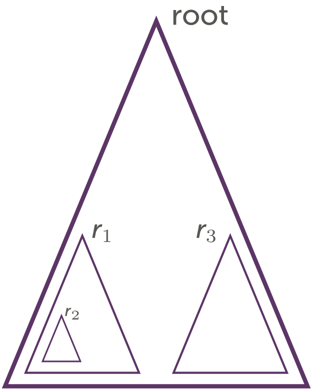
```
let bump (c_in: connection) (c_out: connection): ST unit
  (requires (fun _ ->
    c_in.count <> c_out.count))
  (ensures (fun h_0 _ h_1 ->
    let open U32 in
    Heap.sel h_1 c_in.count =^ Heap.sel h_0 c_in.count +%^ 1ul /\
    Heap.sel h_1 c_out.count =^ Heap.sel h_0 c_out.count +%^ 1ul)
  let open U32 in
  c_in.count := !c_in.count +%^ 1ul;
  c_out.count := !c_out.count +%^ 1ul
```

Why is stateful verification hard? (8)

- Loss of modularity: I need to talk about what I'm doing, but also what I'm not doing
- Quadratic explosion! Need to state anti-aliasing for `c_in.count`, `c_in.nonce`, `c_out.count`, `c_out.nonce`.
- Need to regain compositionality
- That is, separation

Structuring stateful programming with `HyperHeap`

A new memory model: HyperHeaps



"this computation just touches r_2" = non-interference for free

The weakest pre-condition composition leaves other (disjoint) regions untouched.

Coarse-grained separation (1)

Connections now live in a region:

```
type connection (#r: HH.rid) =
| Connection:
    nonce:HH.rref r U32.t ->
    count:HH.rref r U32.t { count <> nonce } ->
    connection #r
```

- Implicit argument for the region
- One connection per-region
- Use a dependent type instead of a record
- Notice the syntax

Coarse-grained separation (2)

```
let nonce (c: connection): ST U32.t
  (requires (fun _ -> True))
  (ensures (fun h_0 _ h_1 -> HH.modifies Set.empty h_0 h_1)) =
  !c.nonce
```

- Not talking about r
- Higher-level modifies predicate talks about the regions

Coarse-grained separation (3)

```
let bump #r_in #r_out
  (c_in: connection #r_in)
  (c_out: connection #r_out):
ST unit
  (requires (fun _ -> r_in <> r_out))
  (ensures (fun h_0 _ h_1 ->
    let open U32 in
    HH.sel h_1 c_in.count =^ HH.sel h_0 c_in.count +%^ 1ul /\
    HH.sel h_1 c_out.count =^ HH.sel h_0 c_out.count +%^ 1ul /\
    HH.modifies
      (Set.union (Set.singleton r_in) (Set.singleton r_out))
      h_0 h_1))
=
  let open U32 in
  c_in.count := !c_in.count +%^ 1ul;
  c_out.count := !c_out.count +%^ 1ul
```

Some notes about the `HyperHeap` memory model (0)

- This still assumes garbage collection
- The regions are for separation purposes only, not memory management
- A library of lemmas facilitate programming

Some notes about the HyperHeap memory model (1)

```
(* A region-id is a pair of a color and unique-id *)
abstract let rid = list (int * int)

(* An rref lives in a given region *)
abstract let rref (id:rid) (a:Type) = Heap.ref a

(* The heap maps region-ids to heaplets *)
type t = Map.t rid heap

(* Ghost operators: two selection operations *)
let sel (#a:Type) (#i:rid) (m:t) (r:rref i a) : GTot a =
  Heap.sel (Map.sel m i) (as_ref r)

let upd (#a:Type) (#i:rid) (m:t) (r:rref i a) (v:a) : GTot t =
  Map.upd m i (Heap.upd (Map.sel m i) (as_ref r) v)
```

Some notes about the HyperHeap memory model (2)

```
(* Note the use of Cons?. *)
abstract val includes : rid -> rid -> GTot bool
let rec includes r1 r2 =
  if r1=r2 then true
  else if List.Tot.length r2 > List.Tot.length r1
  then includes r1 (Cons?.tl r2)
  else false

let disjoint (i:rid) (j:rid) : GTot bool =
  not (includes i j) && not (includes j i)
```

Some notes about the HyperHeap memory model (3)

```
abstract val extends: rid -> rid -> GTot bool
let extends r0 r1 = Cons? r0 && Cons?.tl r0 = r1

abstract val parent: r:rid{r<>root} -> Tot rid
let parent r = Cons?.tl r

let fresh_region (i:rid) (m0:t) (m1:t) =
  (forall j. includes i j ==> not (Map.contains m0 j))
  /\ Map.contains m1 i
```

Some notes about the `HyperHeap` memory model (4)

Some lemmas:

- if two parents are disjoint, two children (**`extends`**) are distinct
- if two parents are disjoint, their two respective fresh children (**`fresh_region`**) are disjoint
- the **`includes`** predicate is transitive
- if a **`parent`** is modified, included children are potentially modified

Some notes about the HyperHeap memory model (5)

```
let modifies_just (s:set rid) (m0:t) (m1:t) =
  Map.equal m1 (Map.concat m1 (Map.restrict (complement s) m0))
  /\ subset (Map.domain m0) (Map.domain m1)

let modifies_one (r:rid) (m0:t) (m1:t) =
  modifies_just (singleton r) m0 m1
```

- Z3 can reason using coarse-grained modifies clauses
- Any heaplet that is not modified is preserved
- Therefore, an un-modified heaplet's references are untouched

Low-Level stateful programming with `HyperStack`

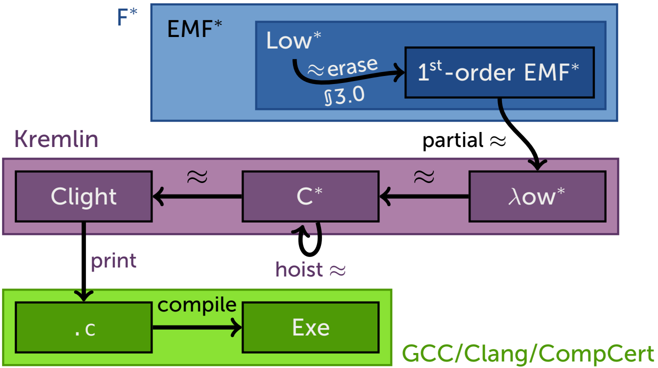
A recap

- Structure is everything!
- Push the `HyperHeap` discipline further, and change the memory model
- Now: a stack of regions, and a heap
- Things are no longer eternal
- Benefit? Extraction to C!

Low*

Low* is a low-level, first-order fragment of F*.

- Offers a limited subset of C's power: stack-allocated buffers and locally mutable variables
- Code is written against a `HyperStack` library
- Suitable pre- and post-conditions ensure memory safety
- If the code $\boxed{\text{ends up}}$ in Low*, it can be translated to C.


F*
EMF*
Low*
≈ erase
§3.0
1st-order EMF*
partial ≈
Kremlin
Clight
≈
C*
≈
λow*
print
hoist ≈
. c
compile
Exe
GCC/Clang/CompCert

The memory model

- A list of stack frames
- The tip is the current stack frame
- Each stack frame maps locations to values
- Special well-parenthesized **push_frame** and **pop_frame**

```
let test1 (_: unit): Stack unit (fun _ -> true) (fun _ _ _ -> true) =
  push_frame ();
  let b = Buffer.create 21l 2ul in
  print_int32 (index b 0ul +%^ index b 1ul);
  pop_frame ()
```

The Stack effect

```
let equal_domains (m0:mem) (m1:mem) =
  m0.tip = m1.tip /\
  Set.equal (Map.domain m0.h) (Map.domain m1.h) /\
  (∀ r. Map.contains m0.h r ==>
    TSet.equal
      (Heap.domain (Map.sel m0.h r))
      (Heap.domain (Map.sel m1.h r)))

effect Stack (a:Type) (pre:st_pre) (post: (mem -> Tot (st_post a))) =
  STATE a (fun (p:st_post a) (h:mem) ->
    pre h /\ (∀ a h1.
      (pre h /\ post h a h1 /\ equal_domains h h1) ==> p a h1))
```

Preserves the layout of the stack and doesn't allocate in any frame.

Liveness

Buffers are C-style arrays.

- passed by reference
- can take an inner pointer (arithmetic)
- the heap is backed by a map; buffers are backed by a sequence

Working with buffers (1)

We model buffers as a sequence.

```
noeq private type _buffer (a:Type) =
  | MkBuffer: max_length:UInt32.t
    -> content:reference (s:seq a{Seq.length s == v max_length})
    -> idx:UInt32.t
    -> length:UInt32.t{v idx + v length <= v max_length}
    -> _buffer a
```

Working with buffers (2): pointer arithmetic

The contents are shared to reason about aliasing.

```
val sub: #a:Type -> b:buffer a
  -> i:UInt32.t{v i + v b.idx < pow2 n}
  -> len:UInt32.t{v i + v len <= length b}
  -> Tot (b':buffer a{b `includes` b' /\ length b' = v len})
let sub #a b i len =
  MkBuffer b.max_length b.content (i +^ b.idx) len
```

Working with buffers (3): reasoning

Just like `ST.get ()`, the `Buffer.as_seq` function grants a pure, ghost view on the buffer.

```
let as_seq #a h (b:buffer a{live h b}):
  GTot (s:seq a{Seq.length s = length b})
=
  Seq.slice (sel h b) (idx b) (idx b + length b)
```

Notice that `as_seq` can only be called in a given heap.

Working with buffers (4): an example

```
let test (): Stack unit (fun _ -> True) (fun _ _ _ -> True) =
  push_frame ();
  assert_norm (16 < pow2 32);
  let b = Buffer.create 0ul 16ul in
  b.(1ul) <- 2ul;
  let h = ST.get () in
  assert (Seq.index (Buffer.as_seq h b) 0 = 0ul);
  pop_frame ()
```

A trickier example

A function in **Stack** requires **push_region** and **pop_region** to allocate. What about code re-use?

```
let test2 (_: unit):
  StackInline (Buffer.buffer Int32.t)
  (requires (fun h0 -> is_stack_region h0.tip))
  (ensures (fun h0 b h1 -> live h1 b /\ Buffer.length b = 2))
=
  let b = Buffer.create 0l 2ul in
  upd b 0ul (C.rand ());
  upd b 1ul (C.rand ());
  b
```

The StackInline effect

```
let inline_stack_inv h h' : GTot Type0 =
  (* The frame invariant is enforced *)
  h.tip = h'.tip
  (* The heap structure is unchanged *)
  /\ Map.domain h.h == Map.domain h'.h
  (* Any region that is not the tip has not seen any allocations *)
  /\ (∀ (r:HH.rid). (r <> h.tip /\ Map.contains h.h r)
      ==> Heap.domain (Map.sel h.h r) == Heap.domain (Map.sel h'.h r))

effect StackInline (a:Type) (pre:st_pre) (post: (mem -> Tot (st_post a))) =
  STATE a (fun (p:st_post a) (h:mem) ->
    pre h /\ (∀ a h1.
      (pre h /\ post h a h1 /\ inline_stack_inv h h1) ==> p a h1))
```

Working with buffers (5): useful predicates

```
(* Modification clauses *)
abstract val modifies_0 h0 h1
abstract val modifies_1 (#a:Type) (b:buffer a) h0 h1

(* Separation *)
val disjoint #a #b (x:buffer a) (y:buffer b): GTot Type0

(* Liveness *)
let live #a (h:mem) (b:buffer a): GTot Type0 =
  contains h b

(* Equality *)
let equal #a h (b:buffer a) h' (b':buffer a): GTot Type0 =
  live h b /\ live h' b' /\ as_seq h b == as_seq h' b'
```

An exercise

- Take the earlier one-time pad example
- Rewrite it in low-level, imperative style
- Use all the lemmas mentioned before.
- Open FStar.Seq, FStar.Buffer!

Skeleton at `http://jonathan.protzenko.fr/mpri/`

Start with memory safety only, then functional spec later

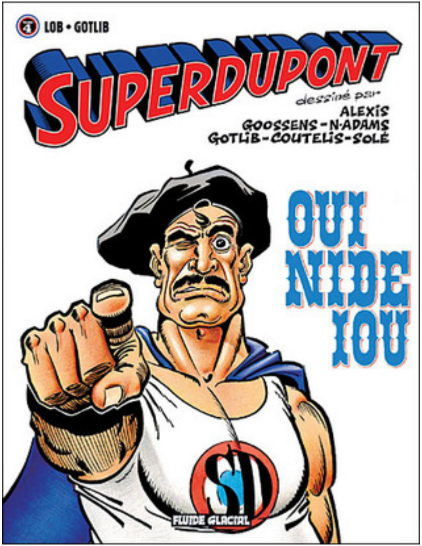
Demo!

```
krml -tmpdir out Ex2Answers2.fst -skip-compilation
```

- All the proofs have been erased
- Only the low-level code remains
- Motto: high-level proofs for low-level code
- We have written 20,000 lines of F* this way

The final word

- We have a bisimulation between λow^* and C^*
- We have a simulation between C^* and C light
- These are all paper proofs
- Research project: rewrite a certified tool in F*



Questions? `jonathan.protzenko@gmail.com`

Thanks for your attention