Program Verification with F*

Danel Ahman

University of Ljubljana, Slovenia

(in January ✈ University of Tartu)

December 6 @ TalTech

About me

- 2007 - 2010, **Tallinn University of Technology**, BSc
- 2011 - 2012, **University of Cambridge**, MPhil (comp. effects)
- 2012 - 2017, **University of Edinburgh**, PhD (eff. & dep. types)

- 2011, **Tallinn University of Technology**, Research Intern
- 2014, **Microsoft Research Silicon Valley**, Research Intern
- 2016, **Microsoft Research Redmond**, Research Intern
- 2019, **Microsoft Research Redmond**, Consulting Researcher

- 2017 - 2018, **Inria Paris**, PostDoc
- 2018 - 2023, **University of Ljubljana**, PostDoc & MSCA Fellow
- 2024 - . . . , **University of Tartu**, Assoc. Prof. of Prog. Langs.

Today's plan

- **Lecture**
  - Crash course in **program verification**
  - What is **F***?
  - Verification of **purely functional** and **stateful programs** in F*
  - Highlights of **other features** of F*
- **Exercise class**
  - Some **interactive live-coding** and **more F* examples**
  - F* applied to writing **verified embedded code for IoT devices**

Today's plan

- **Lecture**
  - Crash course in **program verification**
  - What is **F***?
  - Verification of **purely functional** and **stateful programs** in F*
  - Highlights of **other features** of F*
- **Exercise class**
  - Some **interactive live-coding** and **more F* examples**
  - F* applied to writing **verified embedded code for IoT devices**
- Interested in doing your **MSc dissertation in this area**?
  - Verification of algorithms, protocols, real software, ...
  - Design, meta-theory, and semantics of tools like F*
  - **Get in contact with me and Juhan!**

Specification and verification

Program specification

- Consider a simple **purely functional list reversal**

```
let rec rev #a (l:list a) : list a =
  match l with
  | [] → []
  | hd::tl → append (rev tl) [hd]
```

Program specification

- Consider a simple **purely functional list reversal**

```
let rec rev #a (l:list a) : list a =
  match l with
  | [] → []
  | hd::tl → append (rev tl) [hd]
```

- The **specification of rev** could comprise a variety of properties:
  - rev (rev l) == l
  - length (rev l) == length l
  - rev l contains the same elements as l
  - if sorted ($\geqslant$) l then sorted ($\leqslant$) (rev l)

Program specification

- Consider a simple **purely functional list reversal**

```
let rec rev #a (l:list a) : list a =
  match l with
  | [] → []
  | hd::tl → append (rev tl) [hd]
```

- The **specification of rev** could comprise a variety of properties:
  - rev (rev l) == l
  - length (rev l) == length l
  - rev l contains the same elements as l
  - if sorted (⩾) l then sorted (⩽) (rev l)

- **Specs.** connected to progs. that satisfy them using **Hoare triples**
  - {logical precondition} program {logical postcondition}

Program specification

- Consider a simple **purely functional list reversal**

```
let rec rev #a (l:list a) : list a =
  match l with
  | [] → []
  | hd::tl → append (rev tl) [hd]
```

- The **specification of rev** could comprise a variety of properties:
  - rev (rev l) == l
  - length (rev l) == length l
  - rev l contains the same elements as l
  - if sorted (⩾) l then sorted (⩽) (rev l)

- **Specs.** connected to progs. that satisfy them using **Hoare triples**
  - {logical precondition} program {logical postcondition}
  - {requires (sorted (⩾) l)} rev l {ensures (fun l' → sorted (⩽) l')}

Program specification ctd.

- Now consider a simple **stateful list reversal**

```
let rec srev #a (l:ref (list a)) : unit =
  match !l with
  | [] → ()
  | hd::tl → l := tl; srev tl; l := (append !l [hd])
```

Program specification ctd.

- Now consider a simple **stateful list reversal**

```
let rec srev #a (l:ref (list a)) : unit =
  match !l with
  | [] → ()
  | hd::tl → l := tl; srev tl; l := (append !l [hd])
```

- **Specs.** are still connected to programs using **Hoare triples**
  - specs. now: **previous func. properties** + **memory safety**
  - so they can now also refer to **initial** (h0) and **final** (h1) **heaps**

```
{requires (fun h0 → sorted (≥) (sel h0 l))}
    srev l
{ensures  (fun h0 _ h1 → sorted (≤) (sel h1 l) ∧ modifies !{l} h0 h1)}
```

Program specification ctd.

- Now consider a simple **stateful list reversal**

```
let rec srev #a (l:ref (list a)) : unit =
  match !l with
  | [] → ()
  | hd::tl → l := tl; srev tl; l := (append !l [hd])
```

- **Specs.** are still connected to programs using **Hoare triples**
  - specs. now: **previous func. properties** + **memory safety**
  - so they can now also refer to **initial** (h0) and **final** (h1) **heaps**

```
{requires (fun h0 → sorted (≥) (sel h0 l))}
    srev l
{ensures  (fun h0 _ h1 → sorted (≤) (sel h1 l) ∧ modifies !{l} h0 h1)}
```

- **Memory safety** is important for **composing stateful programs**

```
{requires (fun h0 → sorted (≥) (sel h0 l1) ∧ sorted (≥) (sel h0 l2) ∧ l1 ≠ l2)}
   (srev l1) || (srev l2; srev l2)
{ensures (fun h0 _ h1 → sorted (≤) (sel h1 l1) ∧ sorted (≥) (sel h1 l2))}
```

Program verification

- **Verifying a prog.** is to show that it **satisfies its spec.**, e.g.,

Program verification

- **Verifying a prog.** is to show that it **satisfies its spec.**, e.g.,
  - **Code reviews**

Program verification

- **Verifying a prog.** is to show that it **satisfies its spec.**, e.g.,
  - **Code reviews**
  - **Testing** (unit testing, randomised testing, model-based testing)

Program verification

- **Verifying a prog.** is to show that it **satisfies its spec.**, e.g.,
  - **Code reviews**
  - **Testing** (unit testing, randomised testing, model-based testing)
  - **Runtime verification** (dynamic monitoring)

Program verification

- **Verifying a prog.** is to show that it **satisfies its spec.**, e.g.,
  - **Code reviews**
  - **Testing** (unit testing, randomised testing, model-based testing)
  - **Runtime verification** (dynamic monitoring)
  - **Program logics** (Hoare logic, separation logic, . . .)

Program verification

- **Verifying a prog.** is to show that it **satisfies its spec.**, e.g.,
  - **Code reviews**
  - **Testing** (unit testing, randomised testing, model-based testing)
  - **Runtime verification** (dynamic monitoring)
  - **Program logics** (Hoare logic, separation logic, . . .)

$$\frac{\{\text{fun h0} \to \text{sorted } (\geqslant)\ (\text{sel h0 l})\}\ \text{srev l}\ \{\text{fun h0 \_ h1} \to \text{sorted } (\leqslant)\ (\text{sel h1 l})\} \quad \{\text{fun h1} \to \text{sorted } (\leqslant)\ (\text{sel h1 l})\}\ \text{srev l}\ \{\text{fun h1 \_ h2} \to \text{sorted } (\geqslant)\ (\text{sel h2 l})\}}{\{\text{fun h0} \to \text{sorted } (\geqslant)\ (\text{sel h0 l})\}\ \text{srev l ; srev l}\ \{\text{fun h1 \_ h2} \to \text{sorted } (\geqslant)\ (\text{sel h2 l})\}}$$

Program verification

- **Verifying a prog.** is to show that it **satisfies its spec.**, e.g.,
  - **Code reviews**
  - **Testing** (unit testing, randomised testing, model-based testing)
  - **Runtime verification** (dynamic monitoring)
  - **Program logics** (Hoare logic, separation logic, . . .)

$$\frac{\{\text{fun h0} \to \text{sorted } (\geqslant)\ (\text{sel h0 l})\}\ \text{srev l}\ \{\text{fun h0 \_ h1} \to \text{sorted } (\leqslant)\ (\text{sel h1 l})\} \quad \{\text{fun h1} \to \text{sorted } (\leqslant)\ (\text{sel h1 l})\}\ \text{srev l}\ \{\text{fun h1 \_ h2} \to \text{sorted } (\geqslant)\ (\text{sel h2 l})\}}{\{\text{fun h0} \to \text{sorted } (\geqslant)\ (\text{sel h0 l})\}\ \text{srev l ; srev l}\ \{\text{fun h1 \_ h2} \to \text{sorted } (\geqslant)\ (\text{sel h2 l})\}}$$

  - **Expressive type systems** (dependent types, refinement types)

```
rev : l:list a → (l':list a & (length l == length l'))
```

Program verification

- **Verifying a prog.** is to show that it **satisfies its spec.**, e.g.,
  - **Code reviews**
  - **Testing** (unit testing, randomised testing, model-based testing)
  - **Runtime verification** (dynamic monitoring)
  - **Program logics** (Hoare logic, separation logic, . . .)

$$\frac{\{\text{fun h0} \rightarrow \text{sorted } (\geqslant)\ (\text{sel h0 l})\}\ \text{srev l}\ \{\text{fun h0 \_ h1} \rightarrow \text{sorted } (\leqslant)\ (\text{sel h1 l})\} \quad \{\text{fun h1} \rightarrow \text{sorted } (\leqslant)\ (\text{sel h1 l})\}\ \text{srev l}\ \{\text{fun h1 \_ h2} \rightarrow \text{sorted } (\geqslant)\ (\text{sel h2 l})\}}{\{\text{fun h0} \rightarrow \text{sorted } (\geqslant)\ (\text{sel h0 l})\}\ \text{srev l ; srev l}\ \{\text{fun h1 \_ h2} \rightarrow \text{sorted } (\geqslant)\ (\text{sel h2 l})\}}$$

  - **Expressive type systems** (dependent types, refinement types)

```
rev : l:list a → (l':list a & (length l == length l'))
```

- **F⋆** combines **program logics** with **expressive types** in one sys.!

What is F*?

Prog. verif.: Shall the twain ever meet?

Interactive proof assistants		Semi-automated verifiers of imperative programs
Coq,	*air*	Dafny,
Agda,		FramaC,
Lean,	*gap*	Why3,
Idris		Liquid Haskell

Prog. verif.: Shall the twain ever meet?

Interactive proof assistants		Semi-automated verifiers of imperative programs
Coq,	*air*	Dafny,
Agda,		FramaC,
Lean,	*gap*	Why3,
Idris		Liquid Haskell

- **On the left**
  - very expressive logics, interactive proving, tactics
  - but **mostly only purely functional programming**

Prog. verif.: Shall the twain ever meet?

Interactive proof assistants		Semi-automated verifiers of imperative programs
Coq,	*air*	Dafny,
Agda,		FramaC,
Lean,	*gap*	Why3,
Idris		Liquid Haskell

- **On the left**
  - very expressive logics, interactive proving, tactics
  - but **mostly only purely functional programming**
- **On the right**
  - effectful programming, SMT-based automation
  - but **mostly only very weak logics**

Bridging the air gap: F*

Bridging the air gap: F*

- Developed by Microsoft Research, Inria, and community

Bridging the air gap: F*

- **Developed by Microsoft Research, Inria, and community**
- **Functional programming language with effects**
  - like F#, OCaml, Haskell, ...

```
let incr = fun (r:ref a) → r := !r + 1
```

Bridging the air gap: F*

- **Developed by Microsoft Research, Inria, and community**
- **Functional programming language with effects**
  - like F#, OCaml, Haskell, ...

```
let incr = fun (r:ref a) → r := !r + 1
```

- **By default extracted to OCaml or F#**
  - subsets of F* extracted to efficient C code (using KaRaMeL)

Bridging the air gap: F*

- **Developed by Microsoft Research, Inria, and community**
- **Functional programming language with effects**
  - like F#, OCaml, Haskell, ...

```
let incr = fun (r:ref a) → r := !r + 1
```

- **By default extracted to OCaml or F#**
  - subsets of F* extracted to efficient C code (using KaRaMeL)
- **Semi-automated verification system using SMT (Z3)**
  - **push-button automation** like in Dafny, Why3, Liquid Haskell

Bridging the air gap: F*

- **Developed by Microsoft Research, Inria, and community**
- **Functional programming language with effects**
  - like F#, OCaml, Haskell, ...

```
let incr = fun (r:ref a) → r := !r + 1
```

- **By default extracted to OCaml or F#**
  - subsets of F* extracted to efficient C code (using KaRaMeL)
- **Semi-automated verification system using SMT (Z3)**
  - **push-button automation** like in Dafny, Why3, Liquid Haskell
- **Interactive proof assistant based on dependent types**
  - **interactive proving and tactics** like in Coq, Agda, Idris, Lean

F* in action, at scale

- **A real functional programming language with effects**
  - F* is programmed in F*, but not (yet) verified
- **Project Everest**
  - TLS-1.3 record Layer
  - HACL*, ValeCrypt, and EverCrypt: Cryptographic libraries
  - QUIC record Layer
  - Signal*: A verified secure messaging protocol
  - Vale: A tool for verifying cryptographic primitives in assembly
  - Low* and KaRaMeL: A proof-oriented subset of C embedded in F*
  - EverParse: A parser generator for binary data formats
  - Steel: A concurrent separation logic embedded in F*
  - Pulse: A high-level DSL and typechecker on top of Steel
- For more info, see **fstar-lang.org** and **project-everest.github.io**

How to use F*?

Using F*

- Different kinds of **F* files**
  - A.fsti - interface file for module called A (can be omitted)
  - A.fst - source code file for module called A
  - A.checked, A.hints, ... — generated during typechecking A

Using F*

- Different kinds of **F* files**
  - A.fsti - interface file for module called A (can be omitted)
  - A.fst - source code file for module called A
  - A.checked, A.hints, ... — generated during typechecking A
- **Command line: typechecking/verification**

```
> fstar.exe Ackermann.fst

  Verified module: Ackermann (429 milliseconds)
  All verification conditions discharged successfully
```

Using F*

- Different kinds of **F* files**
  - A.fsti - interface file for module called A (can be omitted)
  - A.fst - source code file for module called A
  - A.checked, A.hints, ... — generated during typechecking A
- **Command line: typechecking/verification**

```
> fstar.exe Ackermann.fst

  Verified module: Ackermann (429 milliseconds)
  All verification conditions discharged successfully
```

- **Command line: typechecking/verif. + program extraction**

```
> fstar.exe Ackermann.fst --odir out-dir --codegen OCaml
```

Using F*

- Different kinds of **F* files**
  - A.fsti - interface file for module called A (can be omitted)
  - A.fst - source code file for module called A
  - A.checked, A.hints, ... — generated during typechecking A
- **Command line: typechecking/verification**

```
> fstar.exe Ackermann.fst

  Verified module: Ackermann (429 milliseconds)
  All verification conditions discharged successfully
```

- **Command line: typechecking/verif. + program extraction**

```
> fstar.exe Ackermann.fst --odir out-dir --codegen OCaml
```

- **Interactive: incremental development + typecheck./verif.**
  - Emacs (fstar-mode) or VS Code (fstar-vscode-assistant)

The core features of F*

The functional core of F*

The functional core of F*

- Recursive functions

```
val factorial : nat → nat

let rec factorial n =
  if n = 0 then 1 else n * (factorial (n − 1))
```

The functional core of F*

- Recursive functions

```
val factorial : nat → nat

let rec factorial n =
  if n = 0 then 1 else n * (factorial (n − 1))
```

- (Simple) inductive datatypes and pattern matching

```
type list (a:Type) =
  | Nil : list a
  | Cons : hd:a → tl:list a → list a

let rec map (f:'a → 'b) (x:list 'a) : list 'b =
  match x with
  | Nil → Nil
  | Cons h t → Cons (f h) (map f t)
```

The functional core of F*

- Recursive functions

```
val factorial : nat → nat

let rec factorial n =
  if n = 0 then 1 else n * (factorial (n − 1))
```

- (Simple) inductive datatypes and pattern matching

```
type list (a:Type) =
  | Nil : list a
  | Cons : hd:a → tl:list a → list a

let rec map (f:'a → 'b) (x:list 'a) : list 'b =
  match x with
  | Nil → Nil
  | Cons h t → Cons (f h) (map f t)
```

- Lambda abstractions

```
map (fun x → x + 42) [1;2;3]
```

Refinement types

- First **advanced feature** compared to OCaml, Haskell, ...

```
type nat = x:int{x ⩾ 0}                    (* general form x:t{phi x} *)
```

  where the formula phi is built from **std. logical connectives**

Refinement types

- First **advanced feature** compared to OCaml, Haskell, ...

```
type nat = x:int{x ⩾ 0}                    (* general form x:t{phi x} *)
```

  where the formula phi is built from **std. logical connectives**

- Refinements **introduced by type annotations** (code unchanged)

```
val factorial : nat → nat
let rec factorial n = if n = 0 then 1 else n * (factorial (n − 1))
```

- Logical obligations **discharged by SMT** (for else branch, simpl.)

```
n ⩾ 0, n <> 0 ⊨ n − 1 ⩾ 0
n ⩾ 0, n <> 0, (factorial (n − 1)) ⩾ 0 ⊨ n * (factorial (n − 1)) ⩾ 0
```

Refinement types

- First **advanced feature** compared to OCaml, Haskell, ...

```
type nat = x:int{x ⩾ 0}                    (* general form x:t{phi x} *)
```

  where the formula phi is built from **std. logical connectives**

- Refinements **introduced by type annotations** (code unchanged)

```
val factorial : nat → nat
let rec factorial n = if n = 0 then 1 else n * (factorial (n − 1))
```

- Logical obligations **discharged by SMT** (for else branch, simpl.)

```
n ⩾ 0, n <> 0 ⊨ n − 1 ⩾ 0
n ⩾ 0, n <> 0, (factorial (n − 1)) ⩾ 0 ⊨ n * (factorial (n − 1)) ⩾ 0
```

- Refinements **eliminated by subtyping**: nat <: int

```
let i : int = factorial 42
let f : x:nat{x > 0} → int = factorial
```

Dependent types

- Next adv. feature is **dependent function types** (aka Π-types)

```
val incr : x:int → y:int{x < y}
let incr x = x + 1
```

Dependent types

- Next adv. feature is **dependent function types** (aka Π-types)

```
val incr : x:int → y:int{x < y}
let incr x = x + 1
```

- Can express **pre- and postconditions** of pure functions

```
val incr' : x:nat{odd x} → y:nat{even y}
```

Dependent types

- Next adv. feature is **dependent function types** (aka Π-types)

```
val incr : x:int → y:int{x < y}
let incr x = x + 1
```

- Can express **pre- and postconditions** of pure functions

```
val incr' : x:nat{odd x} → y:nat{even y}
```

- **Indexed inductive datatypes** and **implicit args.** (#-notation)

```
type vec (a:Type) : nat → Type =
  | Nil : vec a 0
  | Cons : #n:nat → hd:a → tl:vec a n → vec a (n + 1)

let rec vmap (#n:nat) (#a #b:Type) (f:a → b) (as:vec a n) : vec b n =
  match as with
  | Nil → Nil
  | Cons hd tl → Cons (f hd) (vmap f tl)
```

Inductive families + refinement types

- In Coq or Agda, we have to **carry around explicit proofs**, e.g.,

```
type vec (a:Type) : nat → Type =
  | Nil : vec a 0
  | Cons : #n:nat → hd:a → tl:vec a n → vec a (n + 1)

let rec lookup #a #n (as:vec a n) (i:nat) (p:i `less_than` n) : a = ...
```

Inductive families + refinement types

- In Coq or Agda, we have to **carry around explicit proofs**, e.g.,

```
type vec (a:Type) : nat → Type =
  | Nil : vec a 0
  | Cons : #n:nat → hd:a → tl:vec a n → vec a (n + 1)

let rec lookup #a #n (as:vec a n) (i:nat) (p:i `less_than` n) : a = ...
```

- **Combining vec with refinements** is much more convenient

```
let rec lookup #a #n (as:vec a n) (i:nat{i < n}) : a =
  match as with
  | Cons hd tl → if i = 0 then hd else lookup tl (i − 1)
```

Inductive families + refinement types

- In Coq or Agda, we have to **carry around explicit proofs**, e.g.,

```
type vec (a:Type) : nat → Type =
  | Nil : vec a 0
  | Cons : #n:nat → hd:a → tl:vec a n → vec a (n + 1)

let rec lookup #a #n (as:vec a n) (i:nat) (p:i `less_than` n) : a = ...
```

- **Combining vec with refinements** is much more convenient

```
let rec lookup #a #n (as:vec a n) (i:nat{i < n}) : a =
  match as with
  | Cons hd tl → if i = 0 then hd else lookup tl (i − 1)
```

- Often even more convenient to use **simple lists + refinements**

```
let rec length #a (as:list a) : nat = ...

let rec lookup #a (as:list a) (i:nat{i < (length as)}) : a = match as with
  | hd :: tl → if i = 0 then hd else lookup tl (i − 1)

let listvec a n = as:list a{length as = n}
```

Total functions in F* (Tot)

- The F* functions we saw so far were all **total**

Total functions in F* (Tot)

- The F* functions we saw so far were all **total**
- **Tot effect** (default) = no side-effects, terminates on all inputs

```
(* val factorial : nat → nat *)

val factorial : nat → Tot nat

let rec factorial n =
  if n = 0 then 1 else n * (factorial (n − 1))
```

Total functions in F* (Tot)

- The F* functions we saw so far were all **total**
- **Tot effect** (default) = no side-effects, terminates on all inputs

```
(* val factorial : nat → nat *)

val factorial : nat → Tot nat

let rec factorial n =
  if n = 0 then 1 else n * (factorial (n − 1))
```

- **Quiz:** How about giving this weaker type to factorial?

```
val factorial : int → Tot int
```

Total functions in F* (Tot)

- The F* functions we saw so far were all **total**
- **Tot effect** (default) = no side-effects, terminates on all inputs

```
(* val factorial : nat → nat *)

val factorial : nat → Tot nat

let rec factorial n =
  if n = 0 then 1 else n * (factorial (n − 1))
```

- **Quiz:** How about giving this weaker type to factorial?

```
val factorial : int → Tot int
```

```
let rec factorial n = if n = 0 then 1 else n * (factorial (n - 1))
                                                 ^^^^^

Subtyping check failed; expected type (x:int{(x << n)}); got type int
```

Total functions in F* (Tot)

- The F* functions we saw so far were all **total**
- **Tot effect** (default) = no side-effects, terminates on all inputs

```
(* val factorial : nat → nat *)

val factorial : nat → Tot nat

let rec factorial n =
  if n = 0 then 1 else n * (factorial (n − 1))
```

- **Quiz:** How about giving this weaker type to factorial?

```
val factorial : int → Tot int
```

```
let rec factorial n = if n = 0 then 1 else n * (factorial (n - 1))
                                                 ^^^^^

Subtyping check failed; expected type (x:int{(x << n)}); got type int
```

factorial (-1) loops! (int type in F* is unbounded)

The divergence effect (Dv)

- We might **not want to prove all code terminating**

```
val factorial : int → Dv int
```

The divergence effect (Dv)

- We might **not want to prove all code terminating**

```
val factorial : int → Dv int
```

- Some useful code really is **not always terminating**
  - evaluator for lambda terms

```
let rec eval (e : exp) : Dv exp = match e with
  | App (Lam x e1) e2 → eval (subst x e2 e1)
  | App e1 e2 → eval (App (eval e1) e2)
  | Lam x e1 → Lam x (eval e1)
  | _ → e

let main () = eval (App (Lam 0 (App (Var 0) (Var 0)))
                        (Lam 0 (App (Var 0) (Var 0))))
```

The divergence effect (Dv)

- We might **not want to prove all code terminating**

```
val factorial : int → Dv int
```

- Some useful code really is **not always terminating**
  - evaluator for lambda terms

```
let rec eval (e : exp) : Dv exp = match e with
  | App (Lam x e1) e2 → eval (subst x e2 e1)
  | App e1 e2 → eval (App (eval e1) e2)
  | Lam x e1 → Lam x (eval e1)
  | _ → e

let main () = eval (App (Lam 0 (App (Var 0) (Var 0)))
                        (Lam 0 (App (Var 0) (Var 0))))
```

  - servers

```
./webserver.exe
```

The divergence effect (Dv)

- We might **not want to prove all code terminating**

```
val factorial : int → Dv int
```

- Some useful code really is **not always terminating**
  - evaluator for lambda terms

```
let rec eval (e : exp) : Dv exp = match e with
  | App (Lam x e1) e2 → eval (subst x e2 e1)
  | App e1 e2 → eval (App (eval e1) e2)
  | Lam x e1 → Lam x (eval e1)
  | _ → e

let main () = eval (App (Lam 0 (App (Var 0) (Var 0)))
                        (Lam 0 (App (Var 0) (Var 0))))
```

  - servers

```
./webserver.exe
```

  - ...

Effect encapsulation (Tot and Dv)

- Pure code **cannot call potentially divergent code**

Effect encapsulation (Tot and Dv)

- Pure code **cannot call potentially divergent code**
- **Only (!) pure code** can appear **in specifications**

```
val dv_factorial : int → Dv int

type tau = x:int{x = dv_factorial (−1)}
```

Effect encapsulation (Tot and Dv)

- Pure code **cannot call potentially divergent code**
- **Only (!) pure code** can appear **in specifications**

```
val dv_factorial : int → Dv int

type tau = x:int{x = dv_factorial (−1)}
```

```
type tau = x:int{x = dv_factorial (-1)}
                 ^^^^^^^^^^^^^^^^^^^^

Expected a pure expression; got an expression ... with effect "DIV"
```

- **Sub-effecting**: Tot t <: Dv t

Effect encapsulation (Tot and Dv)

- Pure code **cannot call potentially divergent code**
- **Only (!) pure code** can appear **in specifications**

```
val dv_factorial : int → Dv int

type tau = x:int{x = dv_factorial (−1)}
```

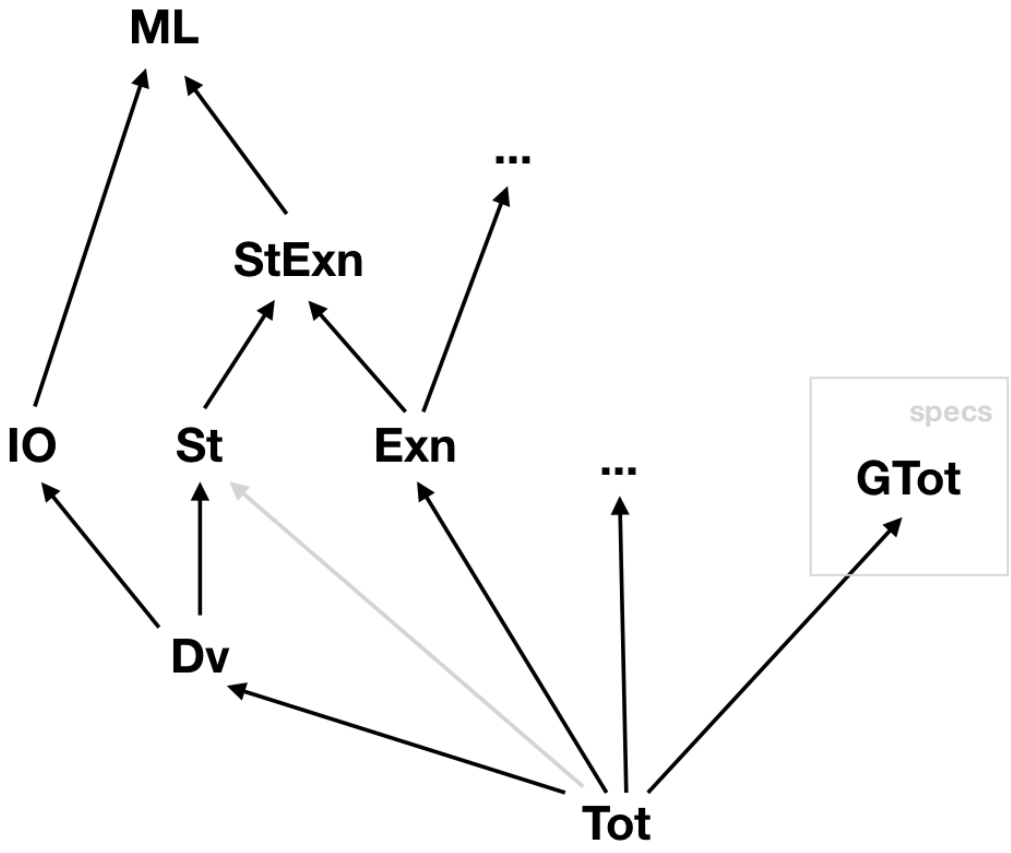
```
type tau = x:int{x = dv_factorial (-1)}
                  ^^^^^^^^^^^^^^^^^^^

Expected a pure expression; got an expression ... with effect "DIV"
```

- **Sub-effecting**: Tot t <: Dv t
- So, **divergent code can include pure code**

```
incr 2 + factorial (−1) : Dv int
```

Tot and Dv are two effs. amongst many



- the **arrows denote the sub-effecting relations** $<:$
- this graph is **user-extensible** (layered effects, Dijkstra monads)

Effect encapsulation (Tot and GTot)

- **Ghost effect** for code used only in specifications

```
val sel : #a:Type → heap → ref a → GTot a
```

Effect encapsulation (Tot and GTot)

- **Ghost effect** for code used only in specifications

```
val sel : #a:Type → heap → ref a → GTot a
```

- **Sub-effecting:** Tot t <: GTot t
- **BUT NOT (!):** GTot t <: Tot t (holds for non-informative ty.)

Effect encapsulation (Tot and GTot)

- **Ghost effect** for code used only in specifications

```
val sel : #a:Type → heap → ref a → GTot a
```

- **Sub-effecting:** Tot t <: GTot t
- **BUT NOT (!):** GTot t <: Tot t (holds for non-informative ty.)
- So, (informative) **ghost code cannot be used** in tot. functions

```
let f (g:unit → GTot nat) : Tot (n:nat{n = g ()}) = g ()
```

```
Computed type "n:nat{n = g ()}" and effect "GTot"
is not compatible with the annotated type "n:nat{n = g ()}" effect "Tot"
```

Effect encapsulation (Tot and GTot)

- **Ghost effect** for code used only in specifications

```
val sel : #a:Type → heap → ref a → GTot a
```

- **Sub-effecting:** Tot t <: GTot t
- **BUT NOT (!):** GTot t <: Tot t (holds for non-informative ty.)
- So, (informative) **ghost code cannot be used** in tot. functions

```
let f (g:unit → GTot nat) : Tot (n:nat{n = g ()}) = g ()
```

```
Computed type "n:nat{n = g ()}" and effect "GTot"
is not compatible with the annotated type "n:nat{n = g ()}" effect "Tot"
```

- But **total functions can be always called** in ghost code

```
let f (g:unit → Tot nat) : GTot (n:nat{n = g ()}) = g ()
```

Effect encapsulation (Tot and GTot)

- **Ghost effect** for code used only in specifications

```
val sel : #a:Type → heap → ref a → GTot a
```

- **Sub-effecting:** Tot t <: GTot t
- **BUT NOT (!):** GTot t <: Tot t (holds for non-informative ty.)
- So, (informative) **ghost code cannot be used** in tot. functions

```
let f (g:unit → GTot nat) : Tot (n:nat{n = g ()}) = g ()
```

```
Computed type "n:nat{n = g ()}" and effect "GTot"
is not compatible with the annotated type "n:nat{n = g ()}" effect "Tot"
```

- Helps to guarantee **code's "irrelevance" of/from specs.**!

```
let f (b:unit → GTot bool) : Tot nat = if b () then ... else ...    (* FAILS TYCHK! *)
```

Verifying purely functional programs in F*

Variant 1: intrinsically (at definition time)

- Using **refinement types** (we saw this already)

```
val factorial : nat → Tot nat                    (* type nat = x:int{x ⩾ 0} *)
```

Variant 1: intrinsically (at definition time)

- Using **refinement types** (we saw this already)

```
val factorial : nat → Tot nat                    (* type nat = x:int{x ⩾ 0} *)
```

- Can equivalently use **pre- and postconditions** for this

```
val factorial : x:int → Pure int (requires (x ⩾ 0))
                                 (ensures  (fun y → y ⩾ 0))
```

  - This is how F* embeds **Hoare logic** for pure programs!

Variant 1: intrinsically (at definition time)

- Using **refinement types** (we saw this already)

```
val factorial : nat → Tot nat                    (* type nat = x:int{x ⩾ 0} *)
```

- Can equivalently use **pre- and postconditions** for this

```
val factorial : x:int → Pure int (requires (x ⩾ 0))
                                 (ensures  (fun y → y ⩾ 0))
```

  - This is how F* embeds **Hoare logic** for pure programs!

- Each F*'s **computation type** is of the form
  - Effect (e.g., Pure) result type (e.g., int) spec. (e.g., pre-post)

Variant 1: intrinsically (at definition time)

- Using **refinement types** (we saw this already)

```
val factorial : nat → Tot nat                    (* type nat = x:int{x ⩾ 0} *)
```

- Can equivalently use **pre- and postconditions** for this

```
val factorial : x:int → Pure int (requires (x ⩾ 0))
                                 (ensures  (fun y → y ⩾ 0))
```

  - This is how F* embeds **Hoare logic** for pure programs!

- Each F*'s **computation type** is of the form
  - Effect (e.g., Pure) result type (e.g., int) spec. (e.g., pre-post)

- Tot is **just an abbreviation** for Pure with trivial spec.

```
Tot t  =  Pure t (requires True) (ensures (fun _ → True))
```

Variant 2: extrinsically (using lemmas)

- **Simply typed definition** of list append

```
let rec append (#a:Type) (xs ys:list a) : Tot (list a) = ...
```

Variant 2: extrinsically (using lemmas)

- **Simply typed definition** of list append

```
let rec append (#a:Type) (xs ys:list a) : Tot (list a) = ...
```

- **Separately proved lemma** that length is linear (uses SMT)

```
let rec lemma_append_length (#a:Type) (xs ys:list a)
  : Pure unit
      (requires True)
      (ensures  (fun _ → length (append xs ys) = length xs + length ys)) =

  match xs with
  | [] → ()                                  (* nil−VC: len (app [] ys) = len [] + len ys *)

  | x :: xs' → lemma_append_length xs' ys
(*                  len (app xs' ys) = len xs' + len ys                                    *)
(* cons−VC:                          ⟹ len (app (x::xs') ys) = len (x::xs') + len ys *)
```

Variant 2: extrinsically (using lemmas)

- **Simply typed definition** of list append

```
let rec append (#a:Type) (xs ys:list a) : Tot (list a) = ...
```

- **Separately proved lemma** that length is linear (uses SMT)

```
let rec lemma_append_length (#a:Type) (xs ys:list a)
  : Pure unit
      (requires True)
      (ensures  (fun _ → length (append xs ys) = length xs + length ys)) =

  match xs with
  | [] → ()                                  (* nil–VC: len (app [] ys) = len [] + len ys *)

  | x :: xs' → lemma_append_length xs' ys
(*                  len (app xs' ys) = len xs' + len ys                                    *)
(* cons–VC:                          ⟹ len (app (x::xs') ys) = len (x::xs') + len ys *)
```

- Convenient **syntactic sugar**: the Lemma effect

```
Lemma property  =  Pure unit (requires True) (ensures (fun _ → property))
```

Lemmas are often unavoidable

```
let snoc l h = append l [h]

let rec rev #a (l:list a) : Tot (list a) =
  match l with
  | [] → []
  | hd::tl → snoc (rev tl) hd
```

```
val lemma_rev_involutive : #a:Type → l:list a → Lemma (rev (rev l) == l)

let rec lemma_rev_involutive (#a:Type) l =
  match l with
  | [] → ()
  | hd::tl → lemma_rev_involutive tl; lemma_rev_snoc (rev tl) hd
```

Lemmas are often unavoidable

```
let snoc l h = append l [h]

let rec rev #a (l:list a) : Tot (list a) =
  match l with
  | [] → []
  | hd::tl → snoc (rev tl) hd
```

```
val lemma_rev_snoc : #a:Type → l:list a → h:a → Lemma (rev (snoc l h) == h::rev l)

let rec lemma_rev_snoc (#a:Type) l h =
  match l with
  | [] → ()
  | hd::tl → lemma_rev_snoc tl h
```

```
val lemma_rev_involutive : #a:Type → l:list a → Lemma (rev (rev l) == l)

let rec lemma_rev_involutive (#a:Type) l =
  match l with
  | [] → ()
  | hd::tl → lemma_rev_involutive tl; lemma_rev_snoc (rev tl) hd
```

Lemmas are unavoidable (but SMT helps)

```
let snoc l h = append l [h]

let rec rev #a (l:list a) : Tot (list a) =
  match l with
  | [] → []
  | hd::tl → snoc (rev tl) hd
```

```
val lemma_rev_snoc : #a:Type → l:list a → h:a → Lemma (rev (snoc l h) == h::rev l)
                                                  [ SMTPat (rev (snoc l h)) ]   (* !!! *)
let rec lemma_rev_snoc (#a:Type) l h =
  match l with
  | [] → ()
  | hd::tl → lemma_rev_snoc tl h
```

```
val lemma_rev_involutive : #a:Type → l:list a → Lemma (rev (rev l) == l)

let rec lemma_rev_involutive (#a:Type) l =
  match l with
  | [] → ()
  | hd::tl → lemma_rev_involutive tl (*; lemma_rev_snoc (rev tl) hd*)
```

Div programs verified only intrinsically

- Using refinement types

```
val factorial : nat → Dv nat
```

- Or the Div computation type (using **pre- and postconditions**)

```
val eval_closed : e:exp → Div exp (requires (closed e))
                                  (ensures  (fun e' → Lam? e' ∧ closed e'))
let rec eval_closed e =
  match e with                          (* notice there is no match case for variables *)
  | App e1 e2 →
      let Lam e1' = eval_closed e1 in
      below_subst_beta 0 e1' e2;
      eval_closed (subst (sub_beta e2) e1')
  | Lam e1 → Lam e1
```

- The Dv effect is also just an abbreviation (just like Tot was)

```
Dv t  =  Div t (requires True) (ensures (fun _ → True))
```

Recap: Functional core of F*

- **Variant of dependent type theory**
  - λ, Π, inductives, matches, universe polymorphism, ...
- **General recursion and semantic termination check**
  - potential non-termination is an effect!
- **Refinements**
  - Refined **value types**
    - `x:t{phi x}`
  - Refined **computation types**
    - `Pure t pre post`
    - `Div t pre post`
  - Refinements comp. and proof irrelevant, discharged by SMT
- **Subtyping and sub-effecting** ($<:$)
- **Standard logical connectives** ($==$, $\wedge$, $\vee$, $\forall$, $\exists$, ...)

Verifying stateful programs in F*

Verifying stateful programs

- The St effect—programming with **garbage-collected references**

```
val incr : r:ref int → St unit

let incr r = r := !r + 1
```

Verifying stateful programs

- The St effect—programming with **garbage-collected references**

```
val incr : r:ref int → St unit

let incr r = r := !r + 1
```

- Hoare logic-style **preconditions** and **postconditions** with ST

```
val incr : r:ref int →
  ST unit (requires (fun h0 → True))
          (ensures (fun h0 _ h1 → sel h1 r == sel h0 r + 1 ∧ modifies !{r} h0 h1))
```

  - **precondition** (requires) is a predicate on initial states
  - **postcond.** (ensures) relates initial states, results, and final states

Verifying stateful programs

- The St effect—programming with **garbage-collected references**

```
val incr : r:ref int → St unit

let incr r = r := !r + 1
```

- Hoare logic-style **preconditions** and **postconditions** with ST

```
val incr : r:ref int →
  ST unit (requires (fun h0 → True))
          (ensures (fun h0 _ h1 → sel h1 r == sel h0 r + 1 ∧ modifies !{r} h0 h1))
```

  - **precondition** (requires) is a predicate on initial states
  - **postcond.** (ensures) relates initial states, results, and final states

- St is again **just an abbreviation**

```
St t  =  ST t (requires True) (ensures (fun _ → True))
```

Verifying stateful programs

- The St effect—programming with **garbage-collected references**

```
val incr : r:ref int → St unit

let incr r = r := !r + 1
```

- Hoare logic-style **preconditions** and **postconditions** with ST

```
val incr : r:ref int →
  ST unit (requires (fun h0 → True))
          (ensures (fun h0 _ h1 → sel h1 r == sel h0 r + 1 ∧ modifies !{r} h0 h1))
```

  - **precondition** (requires) is a predicate on initial states
  - **postcond.** (ensures) relates initial states, results, and final states

- St is again **just an abbreviation**

```
St t  =  ST t (requires True) (ensures (fun _ → True))
```

- **Sub-effecting**: Pure <: ST and Div <: ST (**partial correctness**)

Heap and ST interfaces (much simplified)

```
module Heap
  val heap : Type
  val ref : Type → Type

  val sel : #a:Type → heap → ref a → GTot a
  val addr_of : #a:Type → ref a → GTot nat
  val contains : #a:Type → heap → ref a → Type0

  let modifies (s:FStar.TSet.set nat) (h0 h1 : heap) =
    ∀ a (r:ref a) . (h0 `contains` r ∧ ~(addr_of r `mem` s)) ⟹ sel h1 r == sel h0 r
```

Heap and ST interfaces (much simplified)

```
module ST

  val alloc : #a:Type → init:a →
    ST (ref a) (requires (fun _ → True))
               (ensures (fun h0 r h1 → modifies !{} h0 h1 ∧
                                       sel h1 r == init ∧
                                       fresh r h0 h1))

  val (!) : #a:Type → r:ref a →
    ST a (requires (fun _      → True))
         (ensures (fun h0 x h1 → h0 == h1 ∧
                                 x == sel h0 r))

  val (:=) : #a:Type → r:ref a → v:a →
    ST unit (requires (fun _      → True))
            (ensures (fun h0 _ h1 → modifies !{r} h0 h1 ∧
                                    sel h1 r == v))
```

How does F* verify incr (intuition)?

- The **spec and definition** of incr

```
val incr : r:ref int →
  ST unit (requires (fun _ → True))
          (ensures (fun h0 _ h2 → modifies !{r} h0 h2 ∧ sel h2 r == sel h0 r + 1))

let incr r = r := !r + 1
```

How does F* verify incr (intuition)?

- The **spec and definition** of incr

```
val incr : r:ref int →
  ST unit (requires (fun _ → True))
          (ensures (fun h0 _ h2 → modifies !{r} h0 h2 ∧ sel h2 r == sel h0 r + 1))

let incr r = r := !r + 1
```

1. F*'s internal representation makes **sequencing explicit**

```
let incr r = let x = !r in r := x + 1
```

How does F* verify incr (intuition)?

- The **spec and definition** of incr

```
val incr : r:ref int →
  ST unit (requires (fun _ → True))
          (ensures (fun h0 _ h2 → modifies !{r} h0 h2 ∧ sel h2 r == sel h0 r + 1))

let incr r = r := !r + 1
```

1. F*'s internal representation makes **sequencing explicit**

```
let incr r = let x = !r in r := x + 1
```

2. F* uses the specs. of ! and := to **infer a specification for incr**

```
val incr : r:ref int →
  ST unit
  (requires (fun _ → True))
  (ensures (fun h0 _ h2 → ∃ h1 x. h0 == h1 ∧ x == sel h0 r ∧          (* (!) *)
                          modifies !{r} h1 h2 ∧ sel h2 r == x + 1))  (* (:=) *)
```

How does F* verify incr (intuition)?

- The **spec and definition** of incr

```
val incr : r:ref int →
  ST unit (requires (fun _ → True))
          (ensures (fun h0 _ h2 → modifies !{r} h0 h2 ∧ sel h2 r == sel h0 r + 1))

let incr r = r := !r + 1
```

1. F*'s internal representation makes **sequencing explicit**

```
let incr r = let x = !r in r := x + 1
```

2. F* uses the specs. of ! and := to **infer a specification for incr**

```
val incr : r:ref int →
  ST unit
  (requires (fun _ → True))
  (ensures (fun h0 _ h2 → ∃ h1 x. h0 == h1 ∧ x == sel h0 r ∧          (* (!) *)
                          modifies !{r} h1 h2 ∧ sel h2 r == x + 1))  (* (:=) *)
```

3. SMT-solver checks **if inferred spec. implies user-provided one**

Typing rule for let / sequencing (intuition)

```
val incr : r:ref int →
  ST unit
      (requires (fun _ → True))
      (ensures (fun h0 _ h2 → ∃ h1 x. h0 == h1 ∧ x == sel h0 r ∧          (* (!) *)
                                modifies !{r} h1 h2 ∧ sel h2 r == x + 1))  (* (:=) *)

let incr r =
  let x = !r in
  r := x + 1
```

Typing rule for let / sequencing (intuition)

```
val incr : r:ref int →
  ST unit
      (requires (fun _ → True))
      (ensures (fun h0 _ h2 → ∃ h1 x. h0 == h1 ∧ x == sel h0 r ∧          (* (!) *)
                              modifies !{r} h1 h2 ∧ sel h2 r == x + 1))   (* (:=) *)

let incr r =
  let x = !r in
  r := x + 1
```

```
G |– e1 : ST t1 (requires (fun h0 → pre)) (ensures  (fun h0 x1 h1 → post))

G, x1:t1 |– e2 : ST t2 (requires (fun h1 → ∃ h0 . post))
                       (ensures  (fun h1 x2 h2 → post'))

————————————————————————————————————————————————————————

G |– let x1 = e1 in e2 : ST t2 (requires (fun h0 → pre))
                               (ensures  (fun h0 x2 h2 → ∃ x1 h1 . post ∧ post'))
```

Reference swapping (hand proof sketch)

```
val swap : r1:ref int → r2:ref int →
  ST unit (requires (fun _      → True))
          (ensures (fun h0 _ h3 → modifies !{r1,r2} h0 h3 ∧
                                  sel h3 r2 == sel h0 r1 ∧
                                  sel h3 r1 == sel h0 r2))

let swap r1 r2 =

  let t = !r1 in          (* (P1): ∃ h1 t. h0 == h1 ∧ t == sel h0 r1 *)

  r1 := !r2;              (* (P2): ∃ h2. modifies !{r1} h1 h2 ∧ sel h2 r1 == sel h1 r2 *)

  r2 := t                 (* (P3): modifies !{r2} h2 h3 ∧ sel h3 r2 == t *)
```

Reference swapping (the in-place version)

```
val swap_add_sub : r1:ref int → r2:ref int →
  ST unit (requires (fun _ → addr_of r1 ≠ addr_of r2 ))
          (ensures (fun h0 _ h1 → modifies !{r1,r2} h0 h1 ∧
                                  sel h1 r1 == sel h0 r2 ∧
                                  sel h1 r2 == sel h0 r1))
let swap_add_sub r1 r2 =
  r1 := !r1 + !r2;
  r2 := !r1 − !r2;
  r1 := !r1 − !r2
```

- Correctness of this variant relies on r1 and r2 **not being aliased**
- . . . and on int being **unbounded** (mathematical) integers

But you won't escape inventing invariants

- **Stateful Counting**: $1 + 1 + 1 + 1 + 1 + 1 + \ldots$

```
let rec count_st (n:nat)
  : ST nat (requires (fun _ → True))
           (ensures (fun h0 x h1 → modifies !{} h0 h1 ∧ x == n))
  =
  let r = alloc 0 in
  count_st_aux r n;
  !r
```

But you won't escape inventing invariants

- **Stateful Counting**: $1 + 1 + 1 + 1 + 1 + 1 + \ldots$

```
let rec count_st (n:nat)
  : ST nat (requires (fun _ → True))
           (ensures (fun h0 x h1 → modifies !{} h0 h1 ∧ x == n))
  =
  let r = alloc 0 in
  count_st_aux r n;
  !r
```

- The (recursive) **loop** and its **invariant**

```
let rec count_st_aux (r:ref nat) (n:nat)
  : ST unit (requires (fun _ → True))
            (ensures (fun h0 _ h1 → modifies !{r} h0 h1 ∧
                                    (* to ensure !{} in count_st *)
                                    sel h1 r == sel h0 r + n
                                    (* sel h1 r == n would be wrong *)))
  =
  if n > 0 then (r := !r + 1; count_st_aux r (n − 1)) else ()
```

Summary: Verifying Stateful Programs

- ML-style **garbage-collected references**

```
val heap : Type
val ref : Type → Type

val sel : #a:Type → heap → ref a → GTot a
val addr_of : #a:Type → ref a → GTot nat

val modifies : s:set nat → h0:heap → h1:heap → Type0
```

- St effect for simple **ML-style programming**

```
let incr (r:ref int) : St unit = r := !r + 1
```

- ST effect for **pre- and postcond. based (intrinsic) reasoning**

```
ST unit (requires (fun h0 → True))
        (ensures (fun h0 _ h2 → modifies !{r} h0 h2 ∧ sel h2 r == n))
```

- But **that's not all** there is to F*'s memory models!
  - **monotonicity**, **regions**, **heaps-and-stacks**, **concur. sep. logics**

Highlights of other F* features

F* has an extensible effect system

- In addition to Tot, St, ..., **users can define their own effects**
  - IO, exceptions, nondeterminism, concurrency, ...
  - (layered) combinations of primitive and user-defiend effects
  - custom sub-effecting ($<:$) relations between effects
- **Axiomatically** (PLDI 2013)
  - Only abstract signatures of effect operations
  - Implementation of effects given in code extraction
- **Using Dijkstra monads** (POPL 2017, ICFP 2019)
  - Both computations and specifications are modelled as monads
  - F* infers the spec. using a monad morphism $\theta : T \rightarrow S$
- **Using layered effects** (MSR TR 2021)
  - New effects on top of existing ones (like with monads in Haskell)

F* has an extensible effect system (ex: ND)

```
let m a = list a

let dm (a : Type) (wp : w a) : Type =
  p:(a → Type0) → squash (wp p) → l:(m a){∀ x. memP x l ⇒ p x}

let irepr (a : Type) (wp: w a) = dm a wp

let ireturn (a : Type) (x : a) : irepr a (w_return x) = λ _ _ → [x]
```

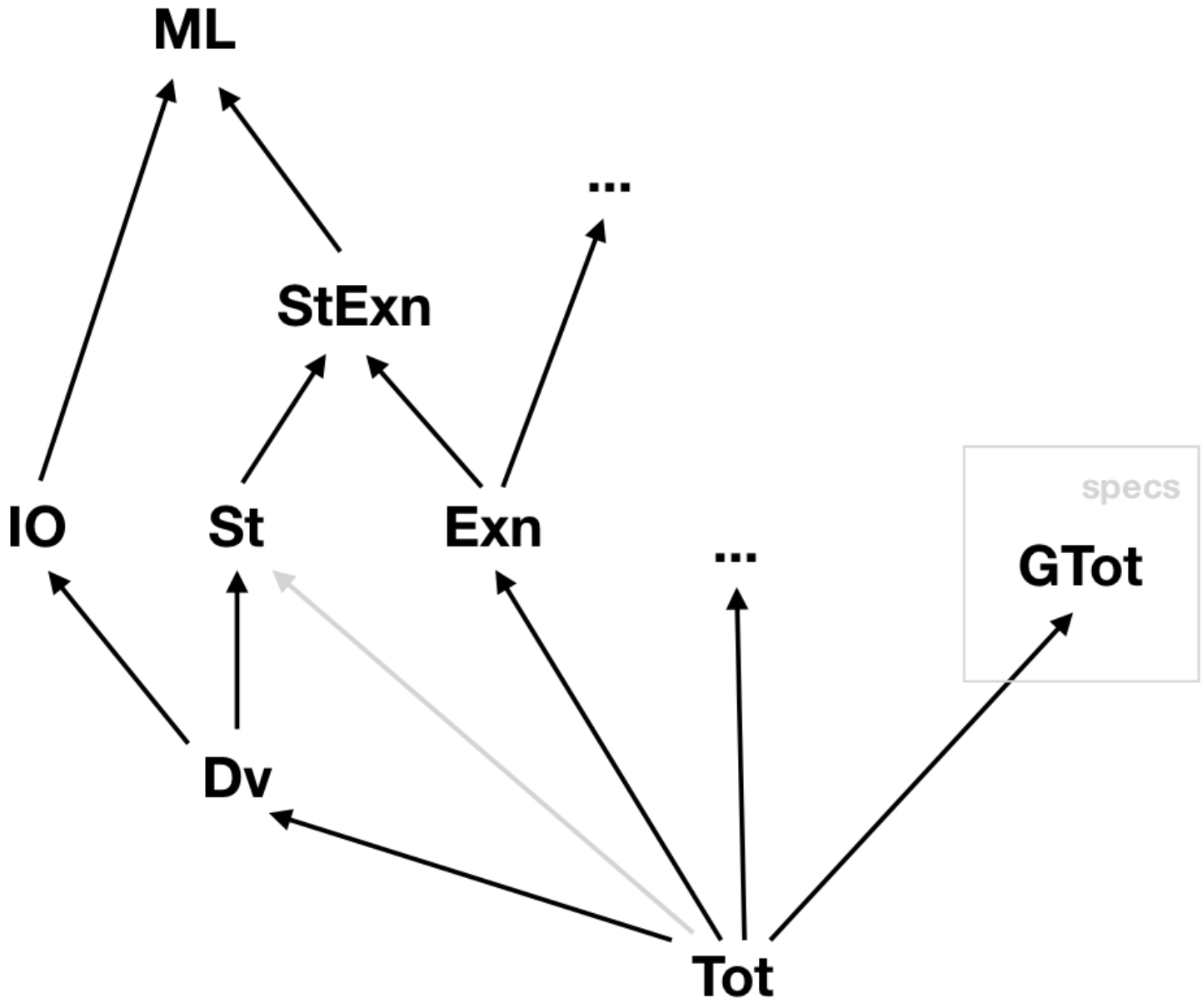
. . .

```
total
reifiable
reflectable
effect {
  ND (a:Type) (wp:w a)
  with {repr = irepr;
        return = ireturn;
        bind = ibind;
        subcomp = isubcomp;
        if_then_else = i_if_then_else}
}

let lift_pure_nd (a:Type) (wp:pure_wp a) (f:unit → PURE a wp) :
  Pure (irepr a wp) (requires ⊤)
                    (ensures (λ _ → ⊤))
  = λ p _ → let r = FStar.Monotonic.Pure.elim_pure f p in [r]

sub_effect PURE ~> ND = lift_pure_nd
```

F* has an extensible effect system (<:)



Low*: verifying low-level C code

- The **code (Low*) is low-level** but **verification (F*) is not**

```
let f (): Stack UInt64.t (requires (fun h0      → True))
                         (ensures (fun h0 r h1 → r == 43UL))

  = push_frame ();                                  (* pushing a new stack frame *)

    let b = LowStar.Buffer.alloca 1UL 64ul in
    assert (b.(42ul) = 1UL);                        (* high—level reasoning in F*'s logic *)

    b.(42ul) ← b.(42ul) +^ 42UL;
    let r = b.(42ul) in

    pop_frame ();                                   (* popping the stack frame we pushed above, *)
                                                    (* necessary for establishing Stack invariant *)

    r
```

Low*: verifying low-level C code

- The **code (Low*) is low-level** but **verification (F*) is not**

```
uint64_t f()
{
  uint64_t b[64U];

  for (uint32_t _i = 0U; _i < (uint32_t)64U; ++_i)
    b[_i] = (uint64_t)1U;

  b[42U] = b[42U] + (uint64_t)42U;
  uint64_t r = b[42U];

  return r;
}
```

- In the lab we will look at some Low* code for **ESP32 IoT device**

Steel: a concurrent separation logic in F*

- Higher-level abstraction on top of SteelCore
  - soundness from an **intrinsically typed definitional interpreter**
  - **commutative monoids (PCM)** based memory model
  - **atomic** and **ghost computations**
  - **dynamically allocated invariants**

```
(* The type of a lock.
   This is implemented as a pair of a boolean reference, and an invariant name *)
val lock (p:vprop) : Type u#0

(* If we have vprop [p] in the context, we can create a new lock
   associated to [p]. [p] is then removed from the context, and only accessible
   through the newly created lock *)
val new_lock (p:vprop)
  : SteelT (lock p) p (λ _ → emp)

(* Acquires the lock, and adds the associated vprop [p] to the context.
   This function loops until the lock becomes available *)
val acquire (#p:vprop) (l:lock p)
  : SteelT unit emp (λ _ → p)

(* Releases a lock [l], as long as the caller previously restored the associated vprop [p] *)
val release (#p:vprop) (l:lock p)
  : SteelT unit p (λ _ → emp)
```

Steel: a concurrent separation logic in F*

- Higher-level abstraction on top of SteelCore
  - soundness from an **intrinsically typed definitional interpreter**
  - **commutative monoids (PCM)** based memory model
  - **atomic** and **ghost computations**
  - **dynamically allocated invariants**

```
val par (#aL:Type u#a)
        (#aR:Type u#a)
        (#preL:vprop)
        (#postL:aL → vprop)
        ($f:unit → SteelT aL preL postL)
        (#preR:vprop)
        (#postR:aR → vprop)
        ($g:unit → SteelT aR preR postR)
  : SteelT (aL & aR)
    (preL `star` preR)
    (λ y → postL (fst y) `star` postR (snd y))
```

Pulse: a DSL for conc. sep. logic in F*

- A **syntax extension** (DSL) and a **custom typechecker** for Steel
- Well-typed Pulse progs. are **extracted to OCaml, C, or Rust**

```pulse
fn fibonacci (k:pos)
  requires emp
  returns r:ℤ
  ensures pure (r == fib k)
{
  let mut i = 1;
  let mut j = 1;
  let mut ctr = 1;
  while ((ctr < k))
  invariant b .
    ∃ vi vj vctr.
    pts_to i vi **
    pts_to j vj **
    pts_to ctr vctr **
    pure (1 ≤ vctr ∧
          vctr ≤ k ∧
          vi == fib (vctr - 1) ∧
          vj == fib vctr ∧
          b == (vctr < k))
  {
      let vi = i;
      ctr := ctr + 1;
      i := j;
      j := vi + j;
  };
  j
}
```

- **Info:** There will be a **Pulse tutorial** (by Nik and others) at POPL

Meta-F*: tactics and metaprogramming

- Tactics are **just another F* effect** (proof state + exceptions)
- Can **access the proof state**
- Can **introspect** and **synthesise F* terms**
- **Run** using the normalizer (slow) or **compiled** to OCaml plugins
- **Uses**:
  - discharging VCs
  - massaging VCs
  - synthesizing terms
  - typeclasses
  - ...

Tactics can discharge verification conditions

```
module Logic

open FStar.Tactics

assume val phi : Type0
assume val psi : Type0
assume val xi  : Type0

let tau () : Tac unit
  = let hyp_phi_xi = implies_intro () in
    right ();
    dump "proofstate 1";
    and_elim (binder_to_term hyp_phi_xi);
    dump "proofstate 2";
    let hyp_phi = implies_intro () in
    let hyp_xi = implies_intro () in
    dump "proofstate 3";
    apply (`FStar.Squash.return_squash);
    dump "proofstate 4";
    exact (binder_to_term hyp_phi);
    qed ()

let _ =
  assert_by_tactic (phi /\ xi ==> psi \/ phi) tau
```

Logic.fst

```
proofstate 1 @ …2018/code/effects/Logic.fst(12,4-12,23) Tue Jul 31
   20:30:07 2018
 Goal 1/1
   hyp_phi_xi: phi /\ xi
   ------------------------------------------------------------
   squash phi
   (*?u46*) _

proofstate 2 @ …2018/code/effects/Logic.fst(14,4-14,23) Tue Jul 31
   20:30:08 2018
 Goal 1/1
   hyp_phi_xi: phi /\ xi
   ------------------------------------------------------------
   squash (phi ==> xi ==> phi)
   (*?u72*) _

proofstate 3 @ …2018/code/effects/Logic.fst(17,4-17,23) Tue Jul 31
   20:30:08 2018
 Goal 1/1
   hyp_phi_xi: phi /\ xi
   hyp_phi: phi
   hyp_xi: xi
   ------------------------------------------------------------
   squash phi
   (*?u121*) _

proofstate 4 @ …2018/code/effects/Logic.fst(19,4-19,23) Tue Jul 31
   20:30:08 2018
 Goal 1/1
   hyp_phi_xi: phi /\ xi
   hyp_phi: phi
   hyp_xi: xi
   ------------------------------------------------------------
   phi
   (*?u130*) _
```

U:--- **Logic.fst** All (22,0) (F✪ FlyC company ElDoc)

U:%%- ***fstar: goals*** Bot (399,0) (F✪ Goals WordWrap)

Tactics can massage verification conditions

CanonCommSemiring.fst

```
assume val modulo_addition_lemma (a:int) (n:pos) (b:int) : Lemma ((a + b * n) % n = a % n)

let lemma_poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh:int) : Lemma
  (requires
    p > 0 /\
    r1 >= 0 /\
    n > 0 /\
    4 * (n * n) == p + 5 /\
    r == r1 * n + r0 /\
    h == h2 * (n * n) + h1 * n + h0 /\
    s1 == r1 + (r1 / 4) /\
    r1 % 4 == 0 /\
    d0 == h0 * r0 + h1 * s1 /\
    d1 == h0 * r1 + h1 * r0 + h2 * s1 /\
    d2 == h2 * r0 /\
    hh == d2 * (n * n) + d1 * n + d0
  )
  (ensures (h * r) % p == hh % p)
  =
  let r14 = r1 / 4 in
  let h_r_expand = (h2 * (n * n) + h1 * n + h0) * ((r14 * 4) * n + r0) in
  let hh_expand = (h2 * r0) * (n * n) + (h0 * (r14 * 4) + h1 * r0 + h2 * (5 * r14)) * n
    + (h0 * r0 + h1 * (5 * r14)) in
  //assert (h * r == h_r_expand);
  //assert (hh == hh_expand);
  let b = ((h2 * n + h1) * r14) in
  modulo_addition_lemma hh_expand p b;
  assert_by_tactic (h_r_expand == hh_expand + b * (n * n * 4 + (-5)))
    (fun _ -> canon_semiring int_cr);
  ()
```

-:--- **CanonCommSemiring.fst** 88% (396,0) Git:master (F✪ FlyC- company ElDoc)

done @ …cs/CanonCommSemiring.fst(324,12-324,23) Tue Jul 31 20:57:25 2018

```
Goal 1/1
  n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh: int
  p: pure_post unit
  uu___: squash (p > 0 /\ r1 >= 0 /\ n > 0 /\ 4 * (n * n) == p + 5 /\ r == r1 * n
        + r0 /\
      h == h2 * (n * n) + h1 * n + h0 /\ s1 == r1 + r1 / 4 /\ r1 % 4 == 0 /\ d0
        == h0 * r0 + h1 * s1 /\
      d1 == h0 * r1 + h1 * r0 + h2 * s1 /\ d2 == h2 * r0 /\ hh == d2 * (n * n) +
        d1 * n + d0 /\
      (forall (pure_result: unit). h * r % p == hh % p ==> p pure_result))
  any_result: int
  uu___: squash (p == any_result)
  pure_result: unit
  uu___: squash (((h2 * r0) * (n * n) + (h0 * ((r1 / 4) * 4) + h1 * r0 + h2 * (5
        * (r1 / 4))) * n +
      (h0 * r0 + h1 * (5 * (r1 / 4))) +
      ((h2 * n + h1) * (r1 / 4)) * p) %
    p =
    ((h2 * r0) * (n * n) + (h0 * ((r1 / 4) * 4) + h1 * r0 + h2 * (5 * (r1 /
      4))) * n +
      (h0 * r0 + h1 * (5 * (r1 / 4)))) %
    p)
--------------------------------------------------------------------------------
  squash (4 * (h2 * (n * (n * (n * (r1 / 4))))) + h2 * (n * (n * r0)) +
      (4 * (n * (n * (h1 * (r1 / 4))))) + n * (h1 * r0)) +
      (4 * (n * (h0 * (r1 / 4)))) + h0 * r0) ==
      h2 * (n * (n * r0)) + (4 * (n * (h0 * (r1 / 4))) + n * (h1 * r0) + 5 * (h2
  * (n * (r1 / 4)))) +
      (h0 * r0 + 5 * (h1 * (r1 / 4))) +
      (4 * (h2 * (n * (n * (n * (r1 / 4))))) + -5 * (h2 * (n * (r1 / 4))) +
      (4 * (n * (n * (h1 * (r1 / 4)))) + -5 * (h1 * (r1 / 4)))))
  (*?u1863*) _

SMT goal 1/1
  n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh: int
  p: pure_post unit
  uu___: squash (p > 0 /\ r1 >= 0 /\ n > 0 /\ 4 * (n * n) == p + 5 /\ r == r1 * n
```

U:%%- ***fstar: goals*** 99% (4882,0) (F✪ Goals WordWrap)

Tactics can synthesize F* terms

Printers.fst

```
let mk_printer dom : Tac unit =
    let nm = match inspect dom with
             | Tv_FVar fv -> inspect_fv fv
             | _ -> fail "not an fv?"
    in
    let nm = maplast (fun s -> s ^ "_print") nm in
    let sv : sigelt_view = Sg_Let false (pack_fv nm) []
                                (mk_printer_type dom)
                                (mk_printer_fun dom) in
    let ses : list sigelt = [pack_sigelt sv] in
    exact (quote ses)

noeq
type t1 =
    | A : int -> string -> t1
    | B : t1 -> int -> t1
    | C : t1
    | D : string -> t1
    | E : t1 -> t1
    | F : (unit -> t1) -> t1

%splice[t1_print] (fun () -> mk_printer (`t1))

let _
  = assert_norm (
      t1_print (A 5 "hey")
      =
      "(Printers.A 5 \"hey\")"
    )

let _
  = assert_norm (
      t1_print (B (D "thing") 42)
      =
      "(Printers.B (Printers.D \"thing\") 42)"
    )
```

-:--- **Printers.fst** 77% (145,0) Git:master (F✪ FlyC company ElDoc)

Printers.t1_print

/Users/danel/FStar/examples/tactics/<dummy>(0,0-0,0)

```
Type
  arg: t1 -> Prims.Tot string

Definition
  visible let  t1_print  : arg: Printers.t1 -> Prims.Tot Prims.string = fun v ->
    let rec ff_rec v_inner =
      (match v_inner with
        | Printers.A a a ->
          FStar.String.concat ""
            [
              "(";
              FStar.String.concat " "
                ["Printers.A"; Printers.print_Prims_int a; Printers.print_Prims_string a];
              ")"
            ]
        | Printers.B a a ->
          FStar.String.concat ""
            ["("; FStar.String.concat " " ["Printers.B"; ff_rec a; Printers.print_Prims_int a]; ")"]
        | Printers.C -> FStar.String.concat " " ["Printers.C"]
        | Printers.D a ->
          FStar.String.concat ""
            ["("; FStar.String.concat " " ["Printers.D"; Printers.print_Prims_string a]; ")"]
        | Printers.E a ->
          FStar.String.concat "" ["("; FStar.String.concat " " ["Printers.E"; ff_rec a]; ")"]
        | Printers.F _ ->
          FStar.String.concat "" ["("; FStar.String.concat " " ["Printers.F"; "?"]; ")"])
      <:
      Prims.string
    in
    ff_rec v
```

U:%%- ***fstar: doc*** All (34,0) (Help)

Tactics can implement typeclasses in F*

Typeclasses.fst

```
(* A class for decidable equality *)
noeq
type deq a = {
  eq    : a -> a -> bool;
  eq_ok : (x:a) -> (y:a) -> Lemma (__fname__eq x y <==> x == y)
}

%splice[eq;eq_ok] (mk_class (`%deq))

(* These methods are generated by the splice *)
(* [@tcnorm] let eq_ok (#a:Type) [|d : deq a|] = d.eq_ok *)
(* [@tcnorm] let eq    (#a:Type) [|d : deq a|] = d.eq *)

(* A way to get `deq a` for any `a : eqtype` *)
let eq_instance_of_eqtype (#a:eqtype) : deq a =
  Mkdeq (fun x y -> x = y) (fun x y -> ())

(* Two concrete instances *)
[@instance] let eq_int : deq int = eq_instance_of_eqtype
[@instance] let eq_bool : deq bool = eq_instance_of_eqtype

(* A few tests *)
let _ = assert (eq 1 1)
let _ = assert (not (eq 1 2))

let _ = assert (eq true true)
let _ = assert (not (eq true false))
```

-:**- **Typeclasses.fst** 9% (48,0) Git-master (F✪ FlyC company ElDoc)

F*

- An ML-style **effectful functional programming language**
- A **semi-automated SMT-based program verifier**
- An **interactive dependently typed proof assistant**
- A **concurrent separation logic**
- Used successfully in **security and crypto verification**

F*

- An ML-style **effectful functional programming language**
- A **semi-automated SMT-based program verifier**
- An **interactive dependently typed proof assistant**
- A **concurrent separation logic**
- Used successfully in **security and crypto verification**

- Contact me and Juhan regarding **MSc dissertation topics**!

F*

- An ML-style **effectful functional programming language**
- A **semi-automated SMT-based program verifier**
- An **interactive dependently typed proof assistant**
- A **concurrent separation logic**
- Used successfully in **security and crypto verification**

- Contact me and Juhan regarding **MSc dissertation topics**!

- **See you in the exercise class** for

a more hands-on experience with F*!