Functional Programming in F*

KC Sivaramakrishnan

Spring 2020

Adapted from VTSA 2019 course on Program Verification with F*

Program verification
Shall the twain ever meet?

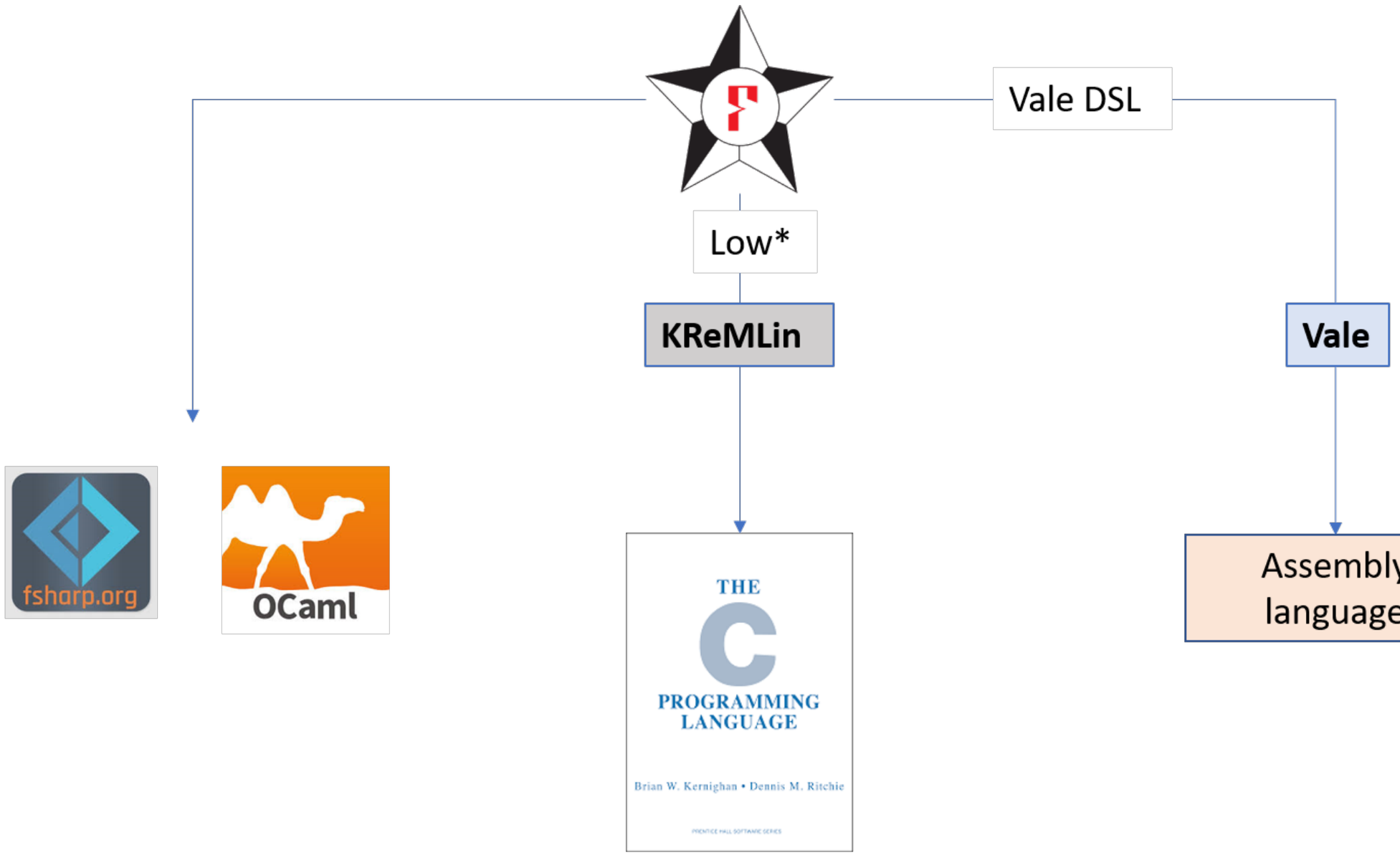
Interactive proof assistants		Semi-automated verifiers of imperative programs
Coq,	*air*	Dafny,
Agda,		FramaC,
Lean,	*gap*	Why3,
Isabelle		Liquid Types

- **Left corner:** very expressive higher-order logics, interactive proving, tactics, but mostly only purely functional programming
- **Right corner:** effectful programming, SMT-based automation, but only very weak logics

Bridging the Gap: F*

- Functional programming language with effects
  - ✦ Like OCaml, Haskell, F#,…
- Semi-automated verification system using SMT
  - ✦ Like Dafny, Frama-C, Why3, Liquid Types,…
- Expressive core language based on dependent type theory
  - ✦ Like Coq, NuPRL, Agda, Lean,…
- A metaprogramming and tactic framework for interactive proof and user-defined automation
  - ✦ Like Coq, NuPRL, Isabelle, HOL, Lean, …

Running F* programs



Uses of F* in Project Everest

- Project Everest: verify and deploy new, efficient HTTPS stack
  - ✦ miTLS: Verified reference implementation of TLS (1.2 and 1.3)
  - ✦ EverParse: Verified parsers and formatter generators
  - ✦ EverCrypt: Agile Cryptographic Provider
  - ✦ HACL*: High-Assurance Cryptographic Library
  - ✦ Vale: Verified Assembly Language for Everest
- Verified Everest code deployed in
  - ✦ Firefox (Mozilla NSS crypto), Windows (WinQUIC), Azure Confidential Consortium (Verified Merkle tree for the blockchain), WireGuard VPN, Zinc crypto library for Linux, Tezos and Concordium blockchains

Switch to Code

Refinement Types

```
type nat = x:int{x >= 0}
```

- 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, simplified)

```
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
```

Switch to Code

Total Functions

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

```
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)

Switch to Code

Semantic Termination Checking

- based on **well-founded ordering on expressions** (<<)
  - naturals related by < (negative integers unrelated)
  - inductives related by subterm ordering
  - lex tuples % [a ; b ; c] with lexicographic ordering
- order constraints discharged by the SMT solver
- arbitrary total expression as **decreases metric**

```
val ackermann: m:nat -> n:nat -> Tot nat (decreases %[m;n])
let rec ackermann m n =
  if m = 0 then n + 1
  else if n = 0 then ackermann (m - 1) 1
  else ackermann (m - 1) (ackermann m (n - 1))
```

- default metric is lex ordering of all (non-function) args

```
val ackermann: m:nat -> n:nat -> Tot nat
```

Switch to Code

Divergence Effect (Dv)

- We might not want to prove that our code always terminates

```
val factorial : int -> Dv int
```

- Some useful code might not always terminate
  - ✦ An evaluator for lambda expressions

```
val eval : exp -> Dv exp
let rec eval e =
  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))))
```

  - ✦ A webserver

- In Coq, we had to write such code as an inductive type and not a Fixpoint

Switch to Code

Effect System

- Pure code cannot call potentially divergent code
- Only pure code can appear in specifications

```
val factorial : int -> Dv int

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

```
type tau = x:int{x = 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

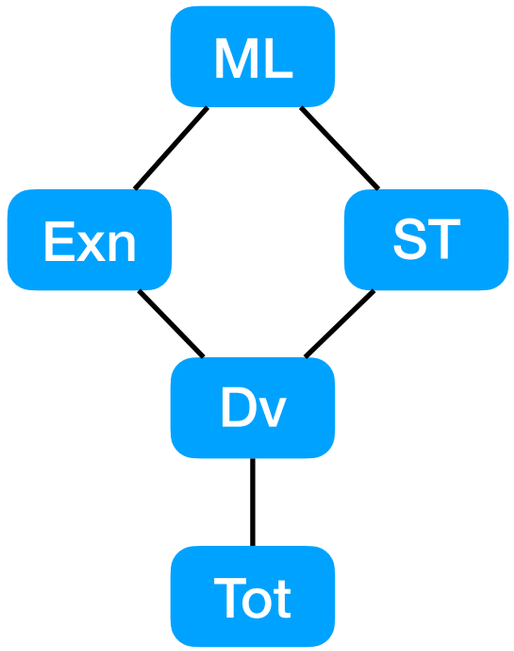
```
incr 2 + factorial (-1) : Dv int
```

Switch to Code

Effect System : Other effects

- Tot and Dv are just two of the possible effects. Some others include:
  - ✦ ST — the effect of a computation that may diverge, read, write or allocate new references in the heap
  - ✦ Exn — the effect of a computation that may diverge or raise an exception
  - ✦ ML — the effect of a computation that may diverge, read, write or allocate, or raise an exception

Effect System: Lattice



- **Sub-effecting:** Computations at a particular effect can be considered to be computations at an effect higher up the lattice

```
let baz () : Dv int = incr 2 + factorial3 (-1)
```

Switch to Code Example

Fin!