# F*, from practice to theory

Kenji Maillard, Ens-Inria Paris

EUTYPES 2018, Nijmegen

# Application-driven development

Everest a verified HTTPS stack,

miTLS a verified implementation of the TLS protocol

HaCL* verified cryptographic primitives

# Application-driven development

| | |
|---|---|
| Everest | a verified HTTPS stack, |
| miTLS | a verified implementation of the TLS protocol |
| HaCL* | verified cryptographic primitives |

- Extraction to OCaml, F# and C
- OCaml, F# : used for compiler, automatic memory management (GC)
- C : used for low-level code, explicit memory management

# Effectful code - Computation types

```
let monotonic_counter () =
  let r = alloc 0 in
  let incr () = r := !r + 1 in
  let get () = !r in
  let result : t = incr, get in result
```

# Effectful code - Computation types

```
type t = (unit → St unit) × (unit → St ℤ)

val monotonic_counter : unit → St t
let monotonic_counter () =
  let r = alloc 0 in
  let incr () = r ≔ !r + 1 <: St unit in
  let get () = !r <: St ℤ in
  let result : t = incr, get in result
```

# Effectful code - Computation types

```
type t = (unit → St unit) × (unit → St ℤ)

val monotonic_counter : unit → St t
let monotonic_counter () =
  let r = alloc 0 in
  let incr () = r := !r + 1 <: St unit in
  let get () = !r <: St ℤ in
  let result : t = incr, get in result
```

Tot total (effect-free) functions

Dv partial functions

St stateful computations

Ex functions throwing exceptions

# Specifying code - pre/post-conditions

```
val fibonacci : n:ℤ → Dv ℤ

let rec fibonacci n =
  if n = 0 ‖ n = 1 then 1
  else fibonacci (n-1) + fibonacci (n-2)
```

# Specifying code - pre/post-conditions

```
val fibonacci : n:ℤ → Pure ℤ
                      (requires _)
                      (ensures _)
let rec fibonacci n =
  if n = 0 ‖ n = 1 then 1
  else fibonacci (n-1) + fibonacci (n-2)
```

# Specifying code - pre/post-conditions

```
val fibonacci : n:ℤ → Pure ℤ
                      (requires (n ≥ 0))
                      (ensures (λ n → n ≥ 1))
let rec fibonacci n =
  if n = 0 || n = 1 then 1
  else fibonacci (n-1) + fibonacci (n-2)
```

# Specifying code - pre/post-conditions

```
val fibonacci : n:ℤ → Pure ℤ
                      (requires (n ≥ 0))
                      (ensures (λ n → n ≥ 1))
let rec fibonacci n =
  if n = 0 || n = 1 then 1
  else fibonacci (n-1) + fibonacci (n-2)
```

- `requires` *precondition*
  for `Pure` ⇝ `precondition : Type`$_0$
- `ensures` *postcondition*
  for `Pure` ⇝ `postcondition : α → Type`$_0$

# Specifying stateful code

```
val incr : r:ref ℤ → St ℤ

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

# Specifying stateful code

```
val incr : r:ref ℤ → ST ℤ
    (requires _)
    (ensures _)
let incr r =
  let x = !r in r := x+1 ; x
```

# Specifying stateful code

```
val incr : r:ref ℤ → ST ℤ
    (requires (λ h₀ → _ ))
    (ensures (λ h₀ x h₁ → _ ))
let incr r =
  let x = !r in r := x+1 ; x
```

- `requires` *precondition*
  for ST ⇝ `precondition : heap → Type₀`
- `ensures` *postcondition*
  for ST ⇝ `postcondition : heap → α → heap → Type₀`

# Specifying stateful code

```
val incr : r:ref ℤ → ST ℤ
    (requires (λ h₀ → T ))
    (ensures (λ h₀ x h₁ → modifies (only r) h₀ h₁
                          ∧ sel h₁ r == x + 1
                          ∧ x = sel h₀ r))
let incr r =
  let x = !r in r := x+1 ; x
```

- requires *precondition*

  for ST ⇝ precondition : heap → $\text{Type}_0$
- ensures *postcondition*

  for ST ⇝ postcondition : heap → α → heap → $\text{Type}_0$

F* by examples

The `Pure` core of F*

Extending F* with monadic effects

# A Dependent Type Theory

The core of F* is a DTT featuring :

- Dependent products and sums

```
  λ x → e : (x:a) → b
(| t, p |) :  (x:a & b)
```

# A Dependent Type Theory

The core of F* is a DTT featuring :

- Dependent products and sums

```
λ x → e : (x:a) → b
(| t, p |) :  (x:a & b)
```

- a hierarchy of predicative universes

```
Type u#n : Type u#(n+1)
```

# A Dependent Type Theory

The core of F* is a DTT featuring :

- Dependent products and sums

```
λ x → e : (x:a) → b
(| t, p |) :  (x:a & b)
```

- a hierarchy of predicative universes

```
Type u#n : Type u#(n+1)
```

- Inductives datatypes & dependent pattern-matching

```
type vector (a:Type) : ℕ → Type =
  | NilV : vector a 0
  | ConsV : (n:ℕ) → a → vector a n → vector a (n+1)
```

# An Extensional Type Theory

Equality reflection & conversion :

$$\textsc{Eq-Refl}\ \frac{\Gamma \vdash \texttt{a == b}}{\Gamma \vdash \mathsf{a} \cong \mathsf{b}} \qquad \textsc{Conv}\ \frac{\Gamma \vdash \texttt{t : a} \qquad \Gamma \vdash \mathsf{a} \cong \mathsf{b}}{\Gamma \vdash t : b}$$

# An Extensional Type Theory

Equality reflection & conversion :

$$\textsc{Eq-Refl}\ \frac{\Gamma \vdash \mathsf{a\ ==\ b}}{\Gamma \vdash \mathsf{a} \cong \mathsf{b}} \qquad \textsc{Conv}\ \frac{\Gamma \vdash \mathtt{t\ :\ a} \qquad \Gamma \vdash \mathsf{a} \cong \mathsf{b}}{\Gamma \vdash t : b}$$

provable equality $\quad$ coincides with $\quad$ definitional equality

a == b $\qquad\qquad$ $\mathsf{a} \cong \mathsf{b}$

- checking whether two terms are equal is undecidable
- As a consequence typechecking is undecidable
- The F* typechecker relies on the SMT to discharge these equalities

# Refinement types

a.k.a. subset types :

$$\texttt{let } \mathbb{N} = \texttt{z}:\mathbb{Z}\{z > 0\}$$

- Introduced by

$$\frac{\Gamma \vdash \texttt{t : a} \qquad \Gamma \vdash \texttt{witness : p[t/x]}}{\Gamma \vdash \texttt{t : (x:a}\{p\}\texttt{)}}$$

- Eliminated by subtyping

$$\Gamma \vdash \texttt{(x:a}\{p\}\texttt{) <: a}$$

# The logical core

Refinement enables defining a squash operation :

```
let squash (p:Type) : Type₀ = x:unit{p}

let prop = a:Type₀{∀ (x:a). x === ()}
```

# The logical core

Refinement enables defining a squash operation :

```
let squash (p:Type) : Type₀ = x:unit{p}

let prop = a:Type₀{∀ (x:a). x === ()}
```

The logical operations are the squashed version of the relevant ones :

```
let ( ∧ ) (p q : Type) : Type₀ = squash (p × q)
let (⟹) (p q : Type) : Type₀ = squash (p → q)

let ( ∀ ) (a:Type) (p:a → Type₀) = squash (x:a → p x)
let ( ∃ ) (a:Type) (p:a → Type₀) = squash (x:a & p x)
```

# Refining computation types

Hoare triples

$$\Gamma \vdash \{\texttt{pre}\}\ \texttt{e}\ \{\texttt{λ (x:a) → post x}\}$$

are classified in F* by the computation type

$$\Gamma \vdash \texttt{e : Pure a pre post}$$

# Refining computation types

Hoare triples

$$\Gamma \vdash \{\texttt{pre}\}\ \texttt{e}\ \{\lambda\ (\texttt{x:a}) \rightarrow \texttt{post x}\}$$

are classified in F* by the computation type

$$\Gamma \vdash \texttt{e : Pure a pre post}$$

which can be recast in the primitive predicate-transformer indexed

$$\Gamma \vdash \texttt{e : PURE a wp}$$

with

```
val wp : (a → Type0) → Type0
let wp (p : a → Type0) = pre ∧ (∀ (x:a). post x ⟹ p x)
```

# Guard condition

A semantic termination criterion is imposed on fixpoints :

```
let rec fibonacci n =
  if n = 0 || n = 1 then 1
  else fibonacci (n-1) + fibonacci (n-2)
```

# Guard condition

A semantic termination criterion is imposed on fixpoints :

```
let rec fibonacci n =
  if n = 0 || n = 1 then 1
  else fibonacci (n-1) + fibonacci (n-2)
```

Inside the recursive definition :

$$\texttt{fibonacci} : \mathtt{n_0}{:}\mathbb{N}\{n_0 \ll n\} \to \mathbb{N}$$

# Guard condition

A semantic termination criterion is imposed on fixpoints :

```
let rec fibonacci n =
  if n = 0 || n = 1 then 1
  else fibonacci (n-1) + fibonacci (n-2)
```

Inside the recursive definition :

$$\texttt{fibonacci} : \texttt{n}_0{:}\mathbb{N}\{n_0 \ll n\} \to \mathbb{N}$$

Generates the verification condition

$$\texttt{(n-1)} \ll \texttt{n} \land \texttt{(n-2)} \ll \texttt{n}$$

# Guard condition

A semantic termination criterion is imposed on fixpoints :

```
let rec fibonacci n =
  if n = 0 || n = 1 then 1
  else fibonacci (n-1) + fibonacci (n-2)
```

Inside the recursive definition :

$$\texttt{fibonacci} : n_0{:}\mathbb{N}\{n_0 \ll n\} \to \mathbb{N}$$

Generates the verification condition

$$(n\text{-}1) \ll n \land (n\text{-}2) \ll n$$

- $n \ll n{+}1$ for positive integer n
- subterm ordering e.g. xs $\ll$ ConsV x xs
- lexicographic ordering $\%[x_1 ; y_1 ; z_1] \ll \%[x_2 ; y_2 ; z_2]$

F* by examples

The `Pure` core of F*

Extending F*with monadic effects

F* by examples

The `Pure` core of F*

Extending F*with monadic effects

# Monadic effects in F*

We build F* effects on top of a monadic specification

```
let mem : Type = ℤ
type st (a:Type) = mem → Tot (a × mem)
```

# Monadic effects in F*

We build F* effects on top of a monadic specification

```
let mem : Type = ℤ
type st (a:Type) = mem → Tot (a × mem)

total new_effect {
  STATE : a:Type → Effect with
  repr = st ;
  return = λ (a:Type) (x:a) (m:mem) → x, m ;
  bind = λ (a b:Type) (f:st a) (g:a → st b) (m:mem) →
           let z,m' = f m in g z m' ;
  get = λ () (m:mem) → m, m;
  put = λ (m0 m1 :mem) → (), m0
}
```

# Monadic effects in F*

We build F* effects on top of a monadic specification

```
let mem : Type = ℤ
type st (a:Type) = mem → Tot (a × mem)

total new_effect {
  STATE : a:Type → Effect with
  repr = st ;
  return = λ (a:Type) (x:a) (m:mem) → x, m ;
  bind = λ (a b:Type) (f:st a) (g:a → st b) (m:mem) →
           let z,m' = f m in g z m' ;
  get = λ () (m:mem) → m, m;
  put = λ (m0 m1 :mem) → (), m0
}
```

- This representation is a model usually kept abstract
- Revealed in specification for reasoning

# Stateful predicate transformer

```
let mem : Type = ℤ
type st_pre = mem → Type₀
type st_post (a:Type) = a × mem → Type₀
type st_wp (a:Type) = st_post a → st_pre
```

# Stateful predicate transformer

```
let mem : Type = ℤ
type st_pre = mem → Type₀
type st_post (a:Type) = a × mem → Type₀
type st_wp (a:Type) = st_post a → st_pre
```

After unfolding and swapping arguments we have :

```
type st_wp (a:Type) = mem → (a × mem → Type₀ ) → Type₀
                    = mem → M (a × mem)
```

where `M X = (X → Type₀) → Type₀`

# Stateful predicate transformer

```
let mem : Type = ℤ
type st_pre = mem → Type₀
type st_post (a:Type) = a × mem → Type₀
type st_wp (a:Type) = st_post a → st_pre
```

After unfolding and swapping arguments we have :

```
type st_wp (a:Type) = mem → (a × mem → Type₀ ) → Type₀
                    = mem → M (a × mem)
```

where `M X = (X → Type₀) → Type₀`

- The weakest-precondition calculus shares some structure with the monadic presentation of the effect

# Dijkstra monads

In order to specify programs effects carries the structure of Dijkstra monad, that is :

# Dijkstra monads

In order to specify programs effects carries the structure of Dijkstra monad, that is :

- A monad of weakest-precondition `WP : Type → Type`

```
let pure_wp a = (a → Type₀) → Type₀
let st_wp a = mem → (a × mem → Type₀) → Type₀
```

# Dijkstra monads

In order to specify programs effects carries the structure of Dijkstra monad, that is :

- A monad of weakest-precondition `WP : Type → Type`

```
let pure_wp a = (a → Type0) → Type0
let st_wp a = mem → (a × mem → Type0) → Type0
```

- a type constructor `T : a:Type → WP a → Type`

```
PURE : (a:Type) → pure_wp a → Type
STATE : (a:Type) → st_wp a → Type
```

# Dijkstra monads

In order to specify programs effects carries the structure of Dijkstra monad, that is :

- A monad of weakest-precondition `WP : Type → Type`

```
let pure_wp a = (a → Type0) → Type0
let st_wp a = mem → (a × mem → Type0) → Type0
```

- a type constructor `T : a:Type → WP a → Type`

```
PURE : (a:Type) → pure_wp a → Type
STATE : (a:Type) → st_wp a → Type
```

- operations indexed by the monad `WP`

# Dijkstra monads - operations

The operations

```
val return : (a:Type) (x:a) → T a (return_wp x)
val bind : (a b:Type) →
      wp₁ :WP a → T a wp₁ →
      (wp₂ :a → WP b) → (x:a → T b (wp₂ x)) →
      T b (bind_wp wp₁ wp₂ )
```

satisfy analogs of the monadic equations, for instance

```
let left_unit (a b:Type) (wp:a → WP b) (f:x:a → T b (wp x)) =
  ∀ (x:a). bind a b (return x) f == f x
```

# Deriving the full STATE effect

```
let st_wp a = mem → (a × mem → Type₀) → Type₀

STATE (a:Type) (wp:st_wp a) =
    m₀ :mem → PURE (a × mem) (wp m₀ <: pure_wp (a × mem))
```

We can elaborate the Dijkstra monad for state on top of PURE !

# Deriving the full STATE effect

```
let st_wp a = mem → (a × mem → Type₀) → Type₀

STATE (a:Type) (wp:st_wp a) =
    m₀:mem → PURE (a × mem) (wp m₀ <: pure_wp (a × mem))
```

We can elaborate the Dijkstra monad for state on top of PURE !

This is not an isolated case :

- We define our effects in DM, a simply typed language

# Deriving the full STATE effect

```
let st_wp a = mem → (a × mem → Type₀) → Type₀

STATE (a:Type) (wp:st_wp a) =
    m₀ :mem → PURE (a × mem) (wp m₀ <: pure_wp (a × mem))
```

We can elaborate the Dijkstra monad for state on top of PURE !

This is not an isolated case :

- We define our effects in DM, a simply typed language
- and generate through DM4Free a djiksta monad in F*

# DM4Free : What do we get ?

- an extensible mechanism for user defined effects such as

  state, exceptions, continuations. . .

# DM4Free : What do we get ?

- an extensible mechanism for user defined effects such as

  state, exceptions, continuations. . .
- a simple semantic, defined on top of the `Pure` core

# DM4Free : What do we get ?

- an extensible mechanism for user defined effects such as

  state, exceptions, continuations. . .
- a simple semantic, defined on top of the `Pure` core
- Compositionality through subeffecting

  $\rightsquigarrow$ monad morphisms elaborate to lifts between effects

# DM4Free : What do we get ?

- an extensible mechanism for user defined effects such as

  state, exceptions, continuations...
- a simple semantic, defined on top of the `Pure` core
- Compositionality through subeffecting

  $\rightsquigarrow$ monad morphisms elaborate to lifts between effects
- reification enables extrinsic reasoning for effectful code

```
val STATE?.reify :
  (unit → STATE a wp) → m₀:mem → GHOST (a × mem) (wp m₀)
```

# Ongoing & Future work

- Extending DM4Free to inductive types/algebraic effects
  $\rightsquigarrow$ Not straightforward, there is no sum of the IO monad with `M X = (X → Type₀) → Type₀` in *Set*
- A categorical account of Dijkstra monads closer to our use
- more generally we strive for a verified metatheory and a (self ?)-certified compiler

# Ongoing & Future work

- Extending DM4Free to inductive types/algebraic effects
  $\rightsquigarrow$ Not straightforward, there is no sum of the IO monad with
  `M X = (X → Type₀) → Type₀` in *Set*
- A categorical account of Dijkstra monads closer to our use
- more generally we strive for a verified metatheory and a (self ?)-certified compiler

# Thank you !

# Semantic of PURE

With DM4Free, the definable computation types are defined on top of PURE.

In order to define a realizability model *à la NuPRL*, we need to give a semantic to PURE, for instance :

$$[\![\texttt{x:a} \to \texttt{PURE b wp}]\!] = \bigcap_{p : b \to \mathbb{P}} x : \{a \mid wp\ p\} \to \{y : b \mid p\ y\}$$

# Semantic of PURE

With DM4Free, the definable computation types are defined on top of PURE.

In order to define a realizability model *à la NuPRL*, we need to give a semantic to PURE, for instance :

$$[\![\texttt{x:a} \rightarrow \texttt{PURE b wp}]\!] = \bigcap_{p:b\rightarrow\mathbb{P}} x : \{a \mid wp\ p\} \rightarrow \{y : b \mid p\ y\}$$

Other obstacles on the way of defining a model :

- $\ll$ is well-founded
- subtyping is coherent
- status of non-terminating functions (does not hinder the logic)

# DM4Free, Graphically

- Two syntactic transfomation from DM to F*
- $e^*$ for specification, $\underline{e}$ for implementation
- Related through typing (logical relation)

$$
\begin{tikzcd}
 & & e^\star : C^\star \arrow[dd, "\text{correctly specifies}"] & \\
(\mathrm{DM}) \quad & e : C \arrow[ur, "\star\text{-translation}"] \arrow[dr, "\text{elaboration}"'] & & \quad (\mathrm{EMF}^\star) \\
 & & \underline{e} : \mathrm{F}_C\ e^\star &
\end{tikzcd}
$$