Towards High-Assurance Cryptographic Software: the F* Proof Assistant

Aymeric Fromherz
Inria Paris,
MPRI 2-30
Outline

• Previously: Proving the security of cryptographic protocols

• Today:
  • Verifying implementations of cryptographic protocols
  • The F* proof assistant
    • The functional core of F*
    • Exercises
  • Try it online at https://fstar-lang.org/tutorial/
  • Or install it locally: https://github.com/FStarLang/FStar
What can go wrong?

Protocol model:

secret \( s \), key \( k \)

\[
\begin{align*}
r & \leftarrow \text{sample}() \\
m & \leftarrow \text{encrypt}(k, \text{concat}(r, s)) \\
\text{send } m
\end{align*}
\]
What can go wrong?

Protocol model:

secret s, key k
r <- sample()
m <- encrypt(k, concat(r, s))
send m

Protocol implementation:

let r = random() in
let m = encrypt(k, r . s) in
send m
What can go wrong?

Protocol model:

```
secret s, key k
r <- sample()
m <- encrypt(k, concat(r, s))
send m
```

Protocol implementation:

```
let random () = 0
let r = random() in
let m = encrypt(k, r . s) in
send m
```
What can go wrong?

Protocol model:

secret s, key k
r <- sample()
m <- encrypt(k, concat(r, s))
send m

Protocol implementation:

print(k)
let r = random() in
let m = encrypt(k, r . s) in
send m
What can go wrong?

Protocol model:

*secret s, key k*

\( r \leftarrow \text{sample}\) \\
\( m \leftarrow \text{encrypt}(k, \text{concat}(r, s))\) \\
send \( m \)

Protocol implementation:

let \( r = \text{random}\) in \\
let \( m = \text{encrypt}(k, r . s)\) in \\
send \( (r . s) \)
A Concrete Example: Modular Arithmetic

• Modular arithmetic is frequently used in cryptographic primitives

\[ g^{xy} \mod p \]

- \( 0 < x, y < p \),
- \( g \) fixed
- \( p \) is frequently a large prime number
  (e.g., \( 2^{255} - 19 \))
Implementing Modular Exponentiation

$$a^b \ mod \ n = a \ * \ a \ * \ ... \ * \ a \ mod \ n$$

• a is a big integer (e.g., $2^{255} - 19$)
• Exponentiation is even bigger
• Machine integers are (at most) 64 bits
• How to implement this? Need a bignum library
Textbook Multiplication

\[ \begin{array}{c}
1101 & 13 \\
\times 1010 & \times 10 \\
- - - - & - - - - \\
= 130
\end{array} \]

\[ \begin{array}{c}
carry \quad 0000 \\
carry + 1101 \\
carry + 0000 \\
+ 1101 \\
- - - - - - \\
10000010
\]
256-bit Modular Multiplication
256-bit Modular Multiplication

What can go wrong?

• Integer overflow (undefined output)
• Buffer overflow/underflow (memory error)
• Missing carry steps (wrong answer)
• Side-Channel attacks (leaks secrets)
Modular Arithmetic Optimizations

• For many primitives, modular arithmetic dominates the crypto overhead
  • $n^2$ 64-bit multiplications
  • Long intermediate arrays
  • Many carry steps

• Many specific optimizations
  • Use only 51 out of 64 bits to reduce carries
  • Precompute reusable intermediate values
  • Use alternative modular reductions (Montgomery, Barrett)
  • Parallelize (vectorize) multiplication and squaring

• Complex optimizations imply more chances of bugs!
Many Bugs in Optimized Bignum Code

[2013] Bug in amd-64-64-24k Curve25519

“Partial audits have revealed a bug in this software (r1 += 0 + carry should be r2 += 0 + carry in amd-64-64-24k) that would not be caught by random tests”
– D.J. Bernstein, W.Janssen, T.Lange, and P.Schwabe

[2014] Arithmetic bug in TweetNaCl’s Curve25519
[2014] Carry bug in Langley’s Donna-32 Curve25519
[2016] Arithmetic bug in OpenSSL Poly1305
[2017] Arithmetic bug in Mozilla NSS GF128
...

TweetNaCL Arithmetic Bug

```c
sv pack25519(u8 *o, const gf n)
{
    int i,j,b;
    gf m,t;
    FOR(i,16) t[i]=n[i];
    car25519(t);
    car25519(t);
    car25519(t);
    FOR(j,2) {
        m[0]=t[0]-0xffed;
        FOR(i=1;i<15;i++) {
            m[i]=t[i]-0xffff-((m[i-1]>>16)&1);
            m[i-1]&=0xffff;
        }n[15]=t[15]-0xffff-((m[14]>>16)&1);
        b=(m[15]>>16)&1;
        n[15]&=0xffff;
        sel25519(t,m,1-b);
    }
    FOR(i,16) {
        o[2*i]=t[i]&0xff;
        o[2*i+1]=t[i]>>8;
    }
}
```

This bug is triggered when the last limb $n[15]$ of the input argument $n$ of this function is greater or equal than $0xffff$. In these cases the result of the scalar multiplication is not reduced as expected resulting in a wrong packed value. This code can be fixed simply by replacing $m[15] &= 0xffff$ by $m[14] &= 0xffff$.

seb.dbzteam.org
Heartbleed (CVE-2014-0160)

• Major vulnerability in OpenSSL TLS implementation
• Affected 17% of all SSL servers
• “Compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users, and the actual content”
• “Allows attackers to eavesdrop on communications, steal data […] and impersonate services and users.”
• Attacks do not leave a trace
Heartbleed (CVE-2014-0160)

- Missing bound check during a memcpy

```c
response = malloc(length);
memcpy(response, recv.heartbeat, length);
```

response = malloc(length);
if length > ssl_state.heartbeat {return 0;}
memcpy(response, recv.heartbeat, length);
GotoFail (CVE-2014-1266)

status SSLVerifyExchange (...) { ...
    if ((err = update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
GotoFail (CVE-2014-1266)

```c
status SSLVerifyExchange (...) { ... 
    if ((err = update(&hashCtx, &signedParams)) != 0) 
        goto fail;
        goto fail;
    if ((err = final(&hashCtx, &hashOut)) != 0) 
        goto fail;
... 
    fail: 
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
} 
```

```c
status SSLVerifyExchange (...) { ... 
    if ((err = update(&hashCtx, &signedParams)) != 0) 
        goto fail;
        goto fail;
    if ((err = final(&hashCtx, &hashOut)) != 0) 
        goto fail;
... 
    fail: 
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
} 
```
GotoFail (CVE-2014-1266)

• Duplicated goto statement in Apple’s TLS implementation
• Bad copy/paste? Faulty merge?

• Impact:
  • Many invalid certificates were accepted
  • Allows using an arbitrary private key for signing or skipping the signing step
  • Enables Man-in-the-Middle attacks

• Many other vulnerabilities: SKIP, FREAK, many memory bugs, correctness issues, infinite loops, ...
Formally Verifying Implementations

- Cryptographic implementations must be correct and secure, but also fast
- Cryptographic implementations are notoriously complex
  - Many tricky optimizations
  - Written in low-level, unsafe languages (C, Assembly)
  - Multiplicity of parameters and variants
- We need strong, formal guarantees about the safety, correctness, and security of cryptographic implementations
The F* Proof Assistant

• A functional programming language
  (like OCaml, Haskell, F#, …)
• With support for dependent types (like Coq, Agda), refinement types, …
• Semi-automated verification by relying on SMT solving
  (like Dafny, Why3, LiquidHaskell, …)
• Also offers a metaprogramming and tactic framework (Meta-F*)
• Extraction to OCaml, F#, C (under certain conditions)

• Try it online at https://fstar-lang.org/tutorial/
• Or install it locally: https://github.com/FStarLang/FStar
F* Applications

- Wide range of applications, mostly security-critical
  - **HACL***: High-Assurance cryptographic library
  - **miTLS**: Verified reference implementation of TLS (1.2 and 1.3)
  - **Noise***: End-to-end verified Implementations of 59 protocols in the Noise family
  - **EverParse**: Verified binary parsers and serializers
  - **StarMalloc**: Verified, concurrent, security-oriented memory allocator
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*

• Inductive types and pattern-matching

```plaintext
type list (a:Type) =
 | Nil : list a
 | Cons : hd: a -> tl: list a -> list a

let rec map (f: a -> b) (l:list a) : list a = match l with
 | [] -> []
 | hd :: tl -> f hd :: map f tl

map (fun x -> x + 3) [1; 2; 3]
```
Dependent Types in F*

• Types can be indexed by values, or other types

```fsharp
val vec (a:Type) : nat -> Type

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

let rec append #a  #n #m (v1: vec a n) (v2: vec a m) : vec a (n + m) =
  match v1 with
  | Nil -> v2
  | Cons hd tl -> Cons hd (append tl v2)
```
Dependent Typechecking

let rec append #a  #n #m (v1: vec a n) (v2: vec a m) : vec a (n + m) =
  match v1 with
  | Nil -> v2
  | Cons hd tl -> Cons hd (append tl v2)

• Two typechecking goals:
  • v1 = Nil |- v2 : vec a (n + m)
  • v1 = Cons hd tl |- Cons hd (append tl v2) : vec a (n + m)

• Case 1: Goal is vec a m = vec a (n + m)
  • v1 = Nil => n = 0. Goal is 0 + m = m.
    Ok by SMT, using F* extensional type theory
Refinement Types

• A *refinement type* is a base type qualified with a logical formula; the formula can express invariants, preconditions, postconditions

• Refinement types are types of the form \( x : T \{ \varphi \} \) where
  • \( T \) is the base type
  • \( x \) refers to the result of the expression, and
  • \( \varphi \) is a logical formula

• The values of this type are the values \( M \) of type \( T \) such that \( \varphi\{M/x\} \) holds
Refinement Types in F*

type nat = n : int \{ n \geq 0 \}

\[ \text{type pos} = n : \text{int} \{ n > 0 \} \]
\[ \text{type neg} = n : \text{int} \{ n < 0 \} \]
\[ \text{type empty} = n : \text{int} \{ \text{False} \} \]

\[ \text{type empty_list (a:Type)} = l : \text{list a} \{ l == [] \} \]
\[ \text{type nonempty_list (a:Type)} = l : \text{list a} \{ l != [] \} \]

let nonempty_hd (l : nonempty_list a) = match l with
  | hd :: _ -> hd

nonempty_hd [1; 2; 3]          // Returns 1
nonempty_hd []                  // Typing error returned by F*
Refinement Subtyping

\[
\text{type } \text{nat} = n : \text{int} \{ n \geq 0 \} \\
\text{type } \text{pos} = n : \text{int} \{ n > 0 \}
\]

• How to ensure that a given integer can be typed as a nat?
  • Ex: 0:int <: nat

• When given an n : pos, how to use it as a n : nat?
  • Ex: 2 : pos <: nat

• We need rules for Refinement Subtyping
Refinement Subtyping: Elimination

• The type \( x : t \{ \varphi \} \) is a subtype of \( t \)
  For any expression \( e : (x : t \{ \varphi \}) \), it is always safe to eliminate the refinement \( \varphi \)

• Examples:
  • \( x : \text{nat} (= \text{int} \{ x \geq 0 \}) <: x : \text{int} \)
  • \( f : \text{list a} -> \text{list a} \), \( l : \text{nonempty_list a} \),
    \( => f l : \text{list a} \)
Refinement Subtyping: Introduction

- For a term $e : t$, $t$ is a subtype of the refinement type $x : t \{ \varphi \}$ if $\varphi[e/x]$.

- Examples:
  - $[x] : \text{nonempty_list a}$
  - If $x : \text{even}$, then $x + 1 : \text{odd}$
Refinement Subtyping

let incr_even (x : even) : odd = x + 1
let incr_odd (x : odd) : even = x + 1

let f (x: int) : int =
  if x % 2 = 0 then incr_even x
  else incr_odd x

If branch, two goals:
• x % 2 = 0 |= x : int <: x : even
• x % 2 = 0 |= incr_even x <: int

Else branch, two goals:
• not (x % 2 = 0) |= x : int <: x : odd
• not (x % 2 = 0) |= incr_odd x <: int
Combining Refinement and Dependent Types

```plaintext
val incr (x:int) : (y:int{y = x + 1})

let incr x = x + 1           // Correctly typechecks
let incr x = x + 2          // Subtyping check failed, expected type y:int{y = x + 1}

val append (#a:Type) (l1 l2:list a) : (l:list a{length l == length l1 + length l2})

val seq_map (#a:Type) (f: a -> a) (s:seq a) : (s’: seq a{
    length s’ == length s ∧
    ∀ (i:nat). i < length s ⇒ s’[i] == f s.[i]})
```
Combining Refinement and Dependent Types

// Sample cryptographic library interface in F*
module AES

type key  // Abstract type for secrets
type block = b: bytes{length b == 16}

val encrypt: k: key -> p:block -> c:block {c == AES(k, p)}
val decrypt: k: key -> c:block -> p:block {c == AES(k, p)}
Type Safety

• Safety means that all logical refinements hold at runtime

• **Theorem (safety):**
  
  For a program $A$ and a type $T$, if $\emptyset \vdash A : T$, then $A$ is safe
Interfaces and Modular Typing

- Interfaces abstract the underlying implementation and definitions
- Interfaces are optional

```fsharp
val seq (a: Type) : Type
val index (#a:Type) (s: seq a) (i:nat{i < length s}) : a
val upd (#a:Type) (s: seq a) (i:nat{i < length s}) (v: a) : seq a
```

```fsharp
let seq (a: Type) = list a
let rec index #a s i =
    if i = 0 then List.hd s else index (List.tl s) (i – 1)
let rec upd #a s i v =
    if i = 0 then v :: List.tl s
    else (List.hd s) :: upd (List.tl s) (i-1) v
```
Modular Typing, Taming Proof Complexity

- Implementation details are not available for verification
- Replacing, e.g., SHA2 by another algorithm does not impact other modules
- Interfaces can be used as abstractions
Modular Typing, Formally

• We write $I_0 \vdash A \sim I$ when, in the typing environment $I_0$, the module $A$ is well-typed and exports the interface $I$

• Theorem (Modular Typing):
  For programs $A_0, A$, interface $I_0$ and type $T$,
  If $\emptyset \vdash A_0 \sim I_0$ and $I_0 \vdash A : T$, then $\emptyset \vdash A_0 . A : T$

• This gives us safety of the program $A_0 . A$ based on the previous theorem
Assertions and Assumptions

Like many other languages, F* supports assertions and assumptions.

• **assert (P)** : Introduce a proof obligation for predicate P
• **assume (P)** : Adds predicate P to the current context.

Examples:

```ml
let f (x : int) : unit =
  assume (x % 2 == 0);
  assert ((x + 1) % 2 == 1)
```

```ml
let f (x : int) : unit =
  assume (False);
  assert (x == x + 1)
```

One can also use **admit ()** to introduce False in the context and admit the remaining of a proof.
Intrinsic vs Extrinsic Verification

• Intrinsic Proof: The type of a term includes properties of interest
  
  \begin{verbatim}
  val list (a:Type) : Type
  val length (#a:Type) (l: list a) : nat

  val append (#a:Type) (l1 l2: list a) : (l: list a{length l == length l1 + length l2})
  \end{verbatim}

• Pros:
  • The proof easily follows the program
  • The property is directly available when calling the function

• Cons:
  • Proving while programming can be tedious
  • The type signature becomes harder to read
  • What about many different properties?
Extrinsic Verification: Lemmas

• F* supports built-in syntax for stating theorems.

val list (a:Type) : Type
val length (#a:Type) (l: list a) : nat
val append (#a:Type) (l1 l2: list a) : list a

val append_length (#a:Type) (l1 l2: list a) :
    Lemma (length l1 + length l2 == length (append l1 l2))
Exercises

• Write the length and append functions, and prove the append_length theorem

• Write a list reverse function, and prove that reverse is involutive

• Write a recursive sum function that sums integers from 1 to n, and prove that it is equal to $\frac{n \cdot (n+1)}{2}$

(You will need the command open FStar.Mul to use the * operator)
F*’s Effect System

• By default, F* functions are total

```latex
let rec factorial (n:nat) : nat =
  if n = 0 then 1 else n * (factorial (n-1))
```
F*’s Effect System

• By default, F* functions are **total**
  
  ```
  let rec factorial (n:nat) : Tot nat =
      if n = 0 then 1 else n * (factorial (n-1))
  ```

• **Tot** is an **effect**, capturing that functions always terminate, and that they have no side-effects.
  
• What happens if we try to give this weaker type to factorial?
  
  ```
  let rec factorial (n:int) : Tot int =
      if n = 0 then 1 else n * (factorial (n-1))
  ```
F* Termination Checker

let rec factorial (n:int) : Tot int =
    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!

Arguments in recursive calls must decrease according to a well-founded ordering <=

Definition: An ordering is well-founded if it does not admit any infinite descending chain
Semantic Termination Checking

• Natural numbers related by < (e.g., 1 << 2 since 1 < 2)
• Inductives related by subterm ordering (e.g., tl << Cons hd tl)
• By default, a recursive function with several arguments uses a lexicographical order on the arguments
Termination Checking, Examples

let rec factorial (n:nat) : Tot nat =
    if n = 0 then 1 else n * (factorial (n-1))

• Goal:  \(n - 1 << n\).
  • Ordering on naturals is \(<\), SMT can prove automatically \(n - 1 < n\)

let rec append #a (l1 l2: list a) : list a =
    match v1 with
    | Nil -> v2
    | Cons hd tl -> Cons hd (append tl v2)

• Goal: \(\%[tl; l2] << \%[l1; l2]\).
  • \(tl << l1\) or \((tl = l1 \land l2 << l2)\)
  • Subterm ordering on \(l1\) gives \(tl << l1\).
Termination Checking, Examples

let rec ackermann (n m:nat) : Tot nat =
  if m=0 then n + 1
  else if n = 0 then ackermann 1 (m - 1)
  else ackermann (ackermann (n - 1) m) (m - 1)

Does this function pass termination checking?
Termination Checking, Examples

let rec ackermann (n m:nat) : Tot nat =
   if m=0 then n + 1
   else if n = 0 then ackermann 1 (m - 1)
   else ackermann (ackermann (n - 1) m) (m - 1)

Does this function pass termination checking?

let rec ackermann (n m:nat) : Tot nat (decreases %[m; n]) =
   if m=0 then n + 1
   else if n = 0 then ackermann 1 (m - 1)
   else ackermann (ackermann (n - 1) m) (m - 1)
F* Effect System: Divergence

• We might want to write non-terminating code:
  • Web servers, operating systems, TLS protocol implementation, ...

• F* provides a built-in *effect* for divergence

```ocaml
definition factorial (n:int) : Dv int =
  if n = 0 then 1 else n * (factorial (n-1))
```

• Code must still typecheck, but termination checker is disabled
Divergence: Avoiding inconsistencies

- Termination is required for consistency in proof assistants
  ```plaintext
  let rec loop () : Dv False = loop ()                           // This typechecks!
  
  let f (x : int) : Tot (y:int{y == x + 1}) = let _ = loop () in x       // What prevents this?
  ```

- F* effect system encapsulates effectful code: By default, different effects cannot interact
  ```plaintext
  let f (x : int) : Tot (y:int{y == x + 1}) = let _ = loop () in x
  ```

  Computed type "int" and effect "DIV" is not compatible with the annotated type "int" effect "Tot"
Subeffecting

• Pure code cannot call potentially divergent code, and only pure code can appear in specifications and proofs.

• But including pure code in divergent code can be useful

  let rec factorial (n:int) : Dv int = if n = 0 then 1 else n * (factorial (n-1))

  We do not want to redefine each basic operator

• F* supports sub-effecting: \texttt{Tot t} \textless= \texttt{Dv t}
Intrinsic Divergence Verification

\[
\text{let rec factorial (n:int) : Dv int =}
\begin{cases}
\text{if } n = 0 & \text{then 1} \\
\text{else } n \times (\text{factorial } (n-1))
\end{cases}
\]

\[
\text{val factorial_lemma (n:int) : Lemma (n} \geq 0 \Rightarrow \text{factorial } n \geq 0)
\]

• Only pure code can appear in specifications

\[
\text{let rec factorial (n:int) : Dv (y:int}\{n} \geq 0 \Rightarrow y \geq 0\} =
\begin{cases}
\text{if } n = 0 & \text{then 1} \\
\text{else } n \times (\text{factorial } (n-1))
\end{cases}
\]

\* \* \*
The GTot effect

• F* also allows writing Ghost code for specifications, proofs, ... which will be erased during extraction.

// Specification of factorial, using natural numbers
val factorial_spec: nat -> GTot nat

// Implementation, using machine integers
val factorial: n:uint64 -> Tot (y:uint64{to_nat y == factorial (to_nat n)})
GTot Subeffecting

• Total code can be used in Ghost functions: Tot t <: GTot t

• Ghost code **cannot** be used in total functions

\[
\begin{align*}
\text{val } f : \text{nat} & \to \text{GTot nat} \\
\text{let } g (n : \text{nat}) : \text{Tot nat} = \\
\text{let } x = f n \text{ in } x + 1
\end{align*}
\]

f is ghost, hence erased at runtime.

How to compile this statement?

• Small subtelty: Ghost code for non-informative types (e.g., ghost values) is allowed (useful for proof purposes)
Refined Computation Types

• So far, refinement in value types:
  
  ```
  val incr (n:int) : Tot (y:int{even n => odd y})
  ```

• F* also allows refined computation types:
  
  ```
  val factorial (n:int) : Pure int (requires n ≥ 0) (ensures fun y -> y ≥ 0)
  ```

• Three elements:
  
  • Effect (here, Pure), result type (here, int), specification (e.g., pre and post)

• Tot t is defined as an abbreviation of
  
  ```
  Pure t (requires True) (ensures fun _ -> True)
  ```
Refined Computation Types

• Other effects are defined in a similar fashion

\[
\text{let rec loop } (_\text{unit}) : \text{Div } \text{unit (requires True) (ensures fun } _ \text{ -> False) = loop ()}
\]

\[
\text{Dv } t \text{ == Div } t \text{ (requires True) (ensures fun } _ \text{ -> True)}
\]

\[
\text{val append_length } (\text{#a:Type}) \text{ (l1 l2: list a) : Ghost } \text{unit}
\]
\[
\text{  (requires True)}
\]
\[
\text{  (ensures fun } _ \text{ -> length l1 + length l2 == length (append l1 l2))}
\]

\[
\text{GTot } t \text{ == Ghost } t \text{ (requires True) (ensures fun } _ \text{ -> True)}
\]

\[
\text{Lemma } (\text{requires P) (ensures Q) = Ghost } \text{unit (requires P) (ensures fun } _ \text{ -> Q)}
\]
Exercises

• Stack, StackClient

• QuickSort: https://fstar-lang.org/tutorial/book/part1/part1_quicksort.html#exercises
Working around the SMT solver

• So far, all F* proofs were discharged by SMT.

• Convenient, automated, but:
  • Cannot reason about induction (manual inductive proofs)
  • Struggles with some theories (e.g., complex modular arithmetic)
  • Performance degrades as the context grows (requires clever abstractions/interfaces for large programs)

• F* provides other reasoning facilities: normalization, the calc statement, and tactics
Proof by Normalization

• Dependently typed proof assistants include a normalizer which reduces computations during typechecking.

• F* provides access to the normalizer for proof purposes.

\[
\text{let rec } \text{length} \, \#a \, (\text{l: list } a) = \text{match } \text{l} \text{ with}
\]
\[
\text{| [] } \to 0 \mid \text{hd :: tl } \to 1 + \text{length tl}
\]

\[
\text{assert} \, (\text{length [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] } = 10)
\]

\[
\text{assert_norm} \, (\text{length [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] } = 10)
\]
Proof by Normalization, Example

```ml
let rec length #a (l: list a) = match l with
    | [] -> 0 | hd :: tl -> 1 + length tl

assert_norm (length [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] == 10)
```

```
match [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] with | [] -> 0 | hd :: tl -> 1 + length tl  == 10  ∼
1 + match [2; 3; 4; 5; 6; 7; 8; 9; 10] with | [] -> 0 | hd :: tl -> 1 + length tl  == 10  ∼
...
10 == 10 ∼
True
```

- Extremely useful for proofs involving recursive functions and concrete terms
Proof by Normalization

• The normalizer only performs reductions, it does not use logical facts in the context

  assert_norm (length [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] == 10) ✔

  let f (l:list a { length l == 10}) = assert_norm (length l == 10) ✗

• The normalizer cannot reduce symbolic terms

• The normalizer can be fine-tuned (only include certain reduction steps, only unfold some definitions, definitions with a given attribute, ...)

63
Calc Statement

• Many (mathematical) proofs consist of a succession of equalities/comparisons:

\[(a + b \cdot 2^c) \cdot 2^d \equiv a \cdot 2^d + b \cdot 2^c \cdot 2^d \equiv a \cdot 2^d + b \cdot 2^{c+d}\]

• F* provides a construct to emulate this:

```plaintext
calc (==) {
    e1;
    (==) { // proof of e1 == e2 }
    e2;
    (==) { // proof of e2 == e3 }
    e3;
}

calc (≥) {
    e1;
    (==) { // proof of e1 == e2 }
    e2;
    (≥) { // proof of e2 ≥ e3 }
    e3;
}
```
F* Tactics

- F* provides a metaprogramming and tactics framework, called Meta-F*
  
  ```plaintext
  assert (pow2 19 == 524288) by (compute (); dump "after compute")
  ```

- Works well for:
  - Small rewritings/goal manipulation
  - Specific types of goals (separation logic, ring normalization)
  - F* goal inspection

- Not recommended as the main proof technique, better to use as a help to SMT
Exercises

• Arithmetic proofs using calc