Extracting from F* to C: a progress report

Peng Wang
MIT, Microsoft Research
Karthikeyan Bhargavan
Jean-Karim Zinzindohoué
INRIA
Abhishek Anand
Cornell University
Cédric Fournet  Bryan Parno  Jonathan Protzenko  Aseem Rastogi  Nikhil Swamy
Microsoft Research

F* is a language in the tradition of ML equipped with dependent types, monadic effects, refinement types and a weakest precondition calculus. Together, these features enable the F* programmer to prove functional correctness using a combination of automation via SMT solving and manual program proofs.

In the context of the greater Everest project, we are using F* to prove, build and deploy miTLS, a verified, efficient implementation of the Transport Layer Security (TLS) 1.3 protocol.

While the current extraction to OCaml may seem sufficient for most use-cases, our ambition is to see our provably-secure code execute in the “real world”; that is, servers (such as Apache, Nginx or IIS) and browsers (such as Chrome, Firefox or Edge). In that context, extracting to OCaml is not an option. The first reason is performance: switching to the OCaml value representation at ABI boundaries, and GC pauses are a hard sell for performance-conscious browser vendors. Second the target audience will most likely not be familiar with OCaml. Thus, for social and technical reasons, our aim is to extract to C or C++.

This extended abstract presents our work in progress. We are currently focusing our efforts on proving the memory safety and functional correctness of Elliptic Curve Cryptography (ECC) primitives, and on extracting this code to C. ECC primitives are a good candidate: the upcoming TLS-1.3 standard makes a significant move towards using ECC in the main ciphersuites. Additionally, crypto routines are extremely performance sensitive and extracting verified implementations to anything other than native code is generally considered unacceptable. As such, we program ECC primitives in a first-order fragment of F*, closely following existing C implementations of those primitives, and after verification extract the code back to C.

The original C code has a straightforward memory management strategy, and only uses stack-based allocation. The ECC primitives are thus written against an F* library that models stacks, and models pushing and popping a new stack frame; suitable pre- and post-conditions ensure that no memory errors can occur. Once this is done, we prove functional correctness of the code (i.e. the math is correct), still using the automation and proving facilities of F*.

Finally, knowing that the F* code has been proven memory safe, we can translate the it back to C, thus obtaining a fully verified C implementation of elliptic curves.

1. Modeling stack-based allocation

Consider the following sample program that exercises our stack library.

```ocaml
open FStar.Int32
open FStar.HyperStack (* for :=, ! and alloc *)

let incr () : STL unit =
  with_frame (fun () ->
    let y = alloc 1ul in
    y := !y + 1ul;
  )
```

Since the original program is shown to perform proper stack-based memory management, the extraction facility need not worry about lifetimes, and can safely translate the original program into the following C code.

```c
void incr()
{
  int32_t y = 1;
  y = y + 1;
}
```

Extraction is sound because:

- the definition of the STL effect guarantees that the caller’s stack is preserved, that is, the function neither pushes more than it pops, nor allocates within its caller’s stack;
- no integer operation overflows thanks to the FStar.Int32.+ operator (this would be undefined C behavior);
- the alloc operation tags y as living within the current stack frame;
- the := and ! operations operate on a stack reference y whose frame (the current one) is still alive.

The stacked memory is modeled on top of hyperheaps. A hyperheap divides memory into nested sub-trees; any function whose effect modifies a given sub-tree can be shown (by virtue of the memory model) to leave any disjoint sub-tree untouched. In short, hyperheaps provide framing guarantees. Each sub-tree is assigned a region-id (rid), and a hyperheap maps an rid to a heap.

A stack of regions is a specific hyperheap that has a list-like structure, starting at the root down to the tip, that is, the top-most stack frame. The is_tip predicate guarantees the list-like shape.
module HH = FStar.HyperHeap

type mem =
| HS : h:HH.t(Map.contains h HH.root /
HH.map_invariant h) -> tip:HH.tid(is_tip tip h) -> mem

val push_frame : unit -> ST unit
  (requires (fun m -> True))
  (ensures (fun (m0:mem) (m1:mem) ->
not (Map.contains m0 h m1.tip) /
HH.parent m1.tip = m0.tip /
  m1.h = Map.upd m0.h m1.tip Heap.emp))

val (!) #a:Type -> r:sref a -> STL a
  (requires (fun m -> is_above r id n tip))
  (ensures (fun s0 v s1 -> s1=s0 /
  v=HyperStack.sel s0 r))

The push_frame combinator allocates a new tip, that is, a
new stack frame, which is initially empty. The (!) combinator requires that the stack reference r live in
a region whose id is equal to, or above, the id of the top-
most stack frame. It ensures that the memory remains unchanged, and it returns the value found at address r.
Other combinators are modeled in a similar manner.

2. Relating F* and C*

We formalize a subset of F* in which effects are represented
à la Haskell, that is, as fully functional transformations that
take state as an input parameter and return an updated
state as an output. This is our reference semantics; the redu-
ation rules only mention expressions. In that context the
state effect is the (imperative, effectful)

state as an output. This is our reference semantics; the re-
formalization in, say, Coq, then the effort required to hook
other languages. Should anyone want to mimic our stack
formalization in, say, Coq, then the effort required to hook
onto the existing tool and get a C backend for free should
be minimal.

Finally, we naturally want to mechanically prove the
soundness of the translation. We hope to perform this task
using F* itself.

If the original program is well-typed in F*, and if it is in
the subset we know how to translate (i.e. uses stack-based
allocation), then we obtain two end-to-end results:

- safety: the resulting C* program will not get stuck;
- preservation of semantics: the extracted C* program
refines the original Low* program.

We reuse a proof strategy in the style of Compcert that
relates Low* and C* through the use of a bisimulation.
More precisely, we prove that C* is a refinement of Low*;
that is, for every step a C* program may take, it corresponds
to a legitimate Low* reduction sequence, hereby justifying
that we have generated a faithful extraction.

Following the Compcert style, we first prove the opposite
direction, that is, Low* is a refinement of C*. Based on
the fact that C* is deterministic, we get the main theorems
above.

3. Implementation and conclusion

F* was equipped last summer with a proper extraction
mechanism in the style of Coq that targets both OCaml
and F#. This extraction mechanism performs the transfor-
mations mentioned earlier, with the exception of A-normal
form transformation, and inner let-binding lifting.

At this stage, the the AST is handed off to an external
tool dubbed KreMLin. KreMLin performs the rest of the
transformations and deals with other mundane matters,
such as avoiding name collisions.

Our ambition for KreMLin is manifold. First, the im-
mediate goal is to extend its input language to handle more
ML-like constructs, such as data types and pairs. Sum types
can be translated using the classic “tagged union” scheme;
records and pairs can be translated using structs. F* does
not have mutable record fields; we can thus pass the latter
structs by value, which means no proof obligations for the
F* programmer.

Second, we hope that the KreMLin tool can prove useful
to other languages. Should anyone want to mimic our stack
formalization in, say, Coq, then the effort required to hook
onto the existing tool and get a C backend for free should
be minimal.

Finally, we naturally want to mechanically prove the
soundness of the translation. We hope to perform this task
using F* itself.

References

[1] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss,
Alfredo Pieroni, Pierre-Yves Strub, and Santiago Zanella-Béguelin. Proving the TLS handshake secure (as it is).
In ACM Symposium on Principles of Programming Languages (POPL’16), 2016.