Modularity, Code Specialization, and Zero-Cost Abstractions for Program Verification

Son Ho (Inria)
Aymeric Fromherz (Inria)
Jonathan Protzenko (Microsoft Research)
Verified Crypto Coming to a Python near you

- Python 3.12
- This work: the story of how we built a layer of high-level APIs
- Technical ingredients: elaborator reflection, meta-programming, automated code rewriting, and high-level abstractions
Background: HACL*

- Integrated in Linux, Firefox, Tezos, and many more
- 140,000+ lines of verified F* code compiling to 80,000+ lines of C
- 30+ algorithms and counting
- Proof engineer productivity is paramount
From Verified Crypto to “Real-World” Software

• HACL* is distributed as **C code**: non-negotiable, for perf.

• We issue a **PR** to “land” new HACL* algorithms **into** a project

• Project owner **reads the generated code**, audits, comments

• Usually, a **back-and-forth** to reach mutual satisfaction

```c
uint32_t hLen = hash_len(a);
KRML_CHECK_SIZE(sizeof(uint8_t), hLen);
uint8_t m1Hash[hLen];
memset(m1Hash, 0U, hLen * sizeof(uint8_t));
uint32_t m1Len = (uint32_t)8U + hLen + saltLen;
KRML_CHECK_SIZE(sizeof(uint8_t), m1Len);
uint8_t m1[m1Len];
memset(m1, 0U, m1Len * sizeof(uint8_t));
memcpy(m1 + (uint32_t)8U, msg, msgLen);
```

Should the patch add a check that `msgLen == hLen`?
The Python Challenge

**hashlib:**
- built-in library of hash functions
- a hodge-podge of implementations, all exposing the same API
- 5 variations of a similar state machine with an internal buffer
- could we factor out this redundancy?

**Can we verify this code generically, and compile it to specialized C code?**

When one thinks of genericity:
- OCaml: functors
- Haskell: typeclasses
- C++: templates
- ...
Encoding Functors: Associative List Example

**OCCaml:**

```ocaml
class Map = sig
type k
val find: k -> (k * 'a) list -> 'a option end

class EqType = sig
type t
val eq: t -> t -> bool end

module MkMap (E : EqType) : Map with type k = E.t = struct
  type k = E.t
  let find x ls =
    let b = ref true in
    let lsp = ref ls in
    while !b do
      match !lsp with
      | [] -> b := false
      | (x', _) :: tl ->
        if E.eq x x' then b := false
        else lsp := tl done;
      match !lsp with
      | [] -> None
      | (_, y) :: _ -> Some y
    end
end
```

**F**: Replace with a linked list

```fsharp
class Map (a : Type) = {
  k: Type;
  find: k -> list (k * a) -> ST (option a) ...
}

class EqType = {
  t: Type;
  eq: t -> t -> bool
}

let mk_map (e : EqType) (a : Type) :
  m: map a{m.k == e.t} = {
  k = e.t;
  find = (fun x ls ->
    let b = alloc true in
    let lsp = alloc ls in
    while (fun () -> !* b) ...
      (fun () ->
        let ls = !* lsp in
        match ls with
        | [] -> upd b false
        | (x', _) :: tl ->
          if e.eq x x' then upd b false
          else upd lsp tl);
      match !lsp with
      | [] -> None
      | (_, y) :: _ -> Some y }
```

Doesn’t compile to C
(same with typeclasses)

Type constraint

Dictionary has runtime cost

Refinement

Proofs and
annotations
omitted

We want a loop in
the generated code

⇒ Specialization and partial evaluation?
Zero-Cost Functors: First Attempt (i)

Generic code (F*):

```ocaml
type map (a : Type) = {
  k: Type;
  find: k -> list (k * a) -> ST (option a) ...
}

type eq_type = {
  t: Type;
  eq: t -> t -> bool;
}

let mk_map (e : eq_type) (a : Type): m:map a{m.k == e.t} = {
  k = e.t;
  find = (fun x ls ->
    let b = alloc true in
    let lsp = alloc ls in
    while (fun () -> !* b)
      (fun () ->
        let ls = !* lsp in
        match ls with
        | [] -> upd b false
        | (x', _) :: tl ->
          if e.eq x x' then upd b false
          else upd lsp tl);
    match !* lsp with
    | [] -> None | (_, y) :: _ -> Some y}
```

Specialization:

```ocaml```

```ocaml
let str_eqty : eq_type = { t = string; eq = String.eq; }
let ifind = (mk_map str_eqty int).find
```

After partial evaluation:  

Types are specialized

```ocaml

let ifind (x: string) (ls: list (string * int)) option int =
  let b = alloc true in let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () ->
      let ls = !* lsp in
      match ls with
      | [] -> upd b false
      | (x', _) :: tl ->
        if String.eq x x' then upd b false
        else upd lsp tl);
    match !* lsp with
    | [] -> None | (_, y) :: _ -> Some y
```

What happens if the code has several layers?
Zero-Cost Functors: First Attempt (ii)

Peer device for a secure channel protocol:

```ocaml
(* "Module signature" *)
let mk_dv (m : map ckey) (c : cipher) : d : dv { d.pid == m.k } = {
    pid = m.k;
    send = (fun id dv plain ->
        match m.find id dv with
        | None -> None
        | Some sk -> Some (c.enc sk plain));
    recv = (fun id dv secret ->
        match m.find id dv with
        | None -> None
        | Some sk -> c.dec sk secret)
}
```
Zero-Cost Functors: Encoding

(* Inline mk_find *)

let mk_find (k v : Type) (eq: k -> k -> bool) (x: k) (ls: list (k * v)) : option v =

  let b = alloc true in let lsp = alloc ts in
  while (fun () -> !* b)
  (fun () -> let ls = !* lsp in
    match ls with | [] -> upd b false
    | (x', _) :: tl -> if eq x x' then upd b false else upd lsp tl);
  match !* lsp with | [] -> None | (_, y) :: _ -> Some y)

(* Don't inline ifind *)

let ifind = mk_find i String.eq

(* Inline mk_send *)

let mk_send (pid : Type) (find : pid -> list (pid * ckey) -> option ckey) (enc : ckey -> bytes -> bytes) (id : pid) (dv : list (pid * ckey)) (plain : bytes) : option bytes =

  match find id dv with
  | None -> None
  | Some sk -> Some (enc sk plain)

(* Don't inline isend *)

let isend = mk_send string ifind aes_enc

... (* mk_recv and irec *)
Zero-Cost Functors: Call-graph Rewriting

What we want to write:

type mindex = { k : Type; v : Type }

[@ Specialize]
assume val eq (i : mindex): i.k -> i.k -> bool

[@ Eliminate]
let while_cond (b: pointer bool) (_:unit) = !*b

[@ Eliminate]
let while_body (i: mindex) (b: pointer bool)
  (lsp: list (i.k * i.v)) (x:i.k) (_:unit) =
  let ls = !* lsp in
  match ls with
  | [] -> upd b false
  | (x', _) :: tl ->
    if eq x x' then upd b false
    else upd lsp tl

[@ Specialize]
let find (i : mindex) (x : i.k)
  (ls : list (i.k * i.v)) : option i.v =
  let b = alloc true in
  let lsp = alloc ls in
  while (while_cond b) (while_body i b lsp x);
  match !* lsp with
  | [] -> None | (_, y) :: _ -> Some y

What we want to get:

type mindex = { k : Type; v : Type }

let mk_find (i: mindex) (eq: i.k-> i.k -> bool)
  (x: i.k) (ls: list (i.k * i.v)): option i.v =
  let b = alloc true in let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () -> let ls = !* lsp in
      match ls with
      | [] -> upd b false
      | (x', _) :: tl ->
        if eq x x' then upd b false
        else upd lsp tl);
  match !* lsp with
  | [] -> None |
  | (_, y) :: _ -> Some y

The code is re-checked

Similar (more complex) device used in the Noise* protocol compiler

%splice [ mk_find ] (specialize (`mindex) [`find ])

Call-graph rewriting by means of meta-programming
Application: algorithms in HACL*

- Type parameter = choice of vectorization level (None, 128-bit, 256-bit)
- Code = crypto algorithm, e.g. Chacha20, Poly1305, etc.
- Deep static call graphs, mixture of [@ Specialize ] and [@ Eliminate ]

```
1 module Hacl.Meta.Chacha20Poly1305
2 #set-options "--z3limit 350 --max_fuel 0 --max_ifuel 1"
3 %splice[
4   mk_chacha20poly1305_poly1305_do;
5   mk_chacha20poly1305_aead_encrypt;
6   mk_chacha20poly1305_aead_decrypt
8   `Hacl.Impl.Chacha20Poly1305.aead_encrypt;
9   `Hacl.Impl.Chacha20Poly1305.aead_decrypt
10  ])
11
12]
13
54 let aead_decrypt : aead_decrypt_st M256 =
55  mk_chacha20poly1305_aead_decrypt #M256 True Hacl.Chacha20.Vec256.chacha20_encrypt_256 poly1305_do_256
```
Application: algorithms in HACL*

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of specializations</th>
<th>Nature of specialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chacha20</td>
<td>3</td>
<td>vectorization level</td>
</tr>
<tr>
<td>Poly1305</td>
<td>3</td>
<td>vectorization level</td>
</tr>
<tr>
<td>Chacha20Poly1305</td>
<td>3</td>
<td>vectorization level</td>
</tr>
<tr>
<td>HPKE</td>
<td>15 (&gt; 80 possible options)</td>
<td>ciphersuite &amp; implementation</td>
</tr>
<tr>
<td>Curve25519</td>
<td>3 (recursive)</td>
<td>field arithmetic</td>
</tr>
</tbody>
</table>

**Curve25519** has two recursive layers of specialization:
- Field64 can be specialized with Vale (ASM) or HACL (C)
- Curve25519 itself can be specialized with Field64 or Field51

All those implementations: > 20k lines of C code
Consider a block API, such as a hash function:

- tricky state machine
- must feed data in entire blocks (unrealistic)
- computing the hash invalidates the state
- precise sequence of operations
Application: Streaming APIs

Instead, people use **Streaming APIs**:  

- Long-lived state carries internal buffer  
- Incremental “update” operation **accumulates** arbitrary-sized data  
- Intermediary digests do not invalidate the state  
- Internal details such as `update_last` are hidden  

- **Tricky to implement correctly**

```c
state *s = hash_new(SHA2_256);
hash_update(s, "hello", 5);
hash_update(s, " ", 1);
char hash1[32];
hash_digest(s, hash1);
char hash2[32];
hash_update(s, "world", 5);
hash_digest(s, hash2);
hash_delete(s);
```
Application: Streaming APIs

A Vulnerability in Implementations of SHA-3, SHAKE, EdDSA, and Other NIST-Approved Algorithms

[CVE-2022-37454] Buffer overflow in the _sha3 module in python versions <= 3.10 #98517
Application: Streaming APIs

• Flagship application of our techniques!

• Use the rewriting tactic and earlier code patterns to write one streaming API that is generic over the choice of block algorithm

• Enormous code savings:
  • 15 applications of the generic code
  • really common API! (though many tweaks: key/ no key, runtime key, etc.)
  • proof-to-code ratio of 0.51: every line of F* yields two lines of C code (total: 8k)
  • in relative terms: massive improvement compared to earlier versions of HACL
Application: Streaming APIs

Extract from one of the “index” types:

```haskell
inline_for_extraction noextract noeq
type block (index: TypeO) =
  Block:
  ks: key_management ->

  // Low-level types
  state: stateful index ->
  key: stateful index ->

  // Just a value type; useful for algorithms that have a variable output length.
  output_length_t: TypeO ->

  // Introducing a notion of blocks and final result.
  max_input_len: (index -> x: U64.t { U64.v x > 0 }) ->
  output_length: (index -> output_length_t -> gtot.Lib.IntTypes.(x:size_int { x > 0 })) ->
  block_len: (index -> x: U32.t { U32.v x > 0 }) ->
  // The size of data to process at a time. Must be a multiple of block_len.
  // Controls the size of the internal buffer.
  blocks_state_len: (i: index -> x: U32.t { U32.v x > 0 } \ U32.v x \= U32.v (block_len i) \ U32.v x \% U32.v (block_len i) = 0 }) ->
  init_input_len: (i: index -> x: U32.t { U32.v x \= U32.v (block_len i) \ U32.v x \% U64.v (max_input_len i) }) ->

  // An init/update/update_last/finish specification. The long refinements were
  // previously defined as blocks / small / output.
  init_state: (i: index -> (key: TypeO) -> s: S.seq uint8 (S.length s = U32.v (init_input_len i) ) ->
  update_state: (i: index -> state: TypeO) -> (key: TypeO) ->
  state: TypeO) ->
  prevlen: (prevlen: U32.v (block_len i) = 0 ) ->
  s: S.seq uint8 (prevlen \+ S.length s \= U64.v (max_input_len i) \ S.length s \% U32.v (block_len i) = 0 ) ->
  state: TypeO) ->
  update_last_state: (i: index -> state: TypeO) ->
  state: TypeO) ->
  prevlen: (prevlen: U32.v (block_len i) = 0 ) ->
  s: S.seq uint8 (S.length s \= prevlen \= U64.v (max_input_len i) \ S.length s \= U32.v (block_len i) ) ->
  state: TypeO) ->
  finish: (i: index -> key: TypeO) -> state: TypeO) ->
  output_length: (output_length_t -> (s: S.seq uint8 (S.length s = output_length i)) ->
```

Types

Constants

Spec (spec defs + theorems)
Application: Streaming APIs

• Excellent engagement with the Python team
• Replaced all of their built-in hash implementations with our verified code
• Released in Python 3.12 (blake2 is coming)
• Good confirmation that our work has practical impact
• Forced us to polish, attain a high level of quality, and do serious packaging work
Modularity, Code Specialization, and Zero-Cost Abstractions for Program Verification

Son Ho (INRIA), Aymeric Fromherz (INRIA), Jonathan Protzenko (Microsoft Research)

- An arsenal of PL techniques to reconcile high-level, generic programming with low-level code specialization and verification (“best of both worlds”)
- Added an extra compiler stage that automatically rewrites the user’s code in userland
- Wide variety of applications in HACL, significant boost on productivity, maintenance
- One flagship application: streaming functor, an “API transformer” that goes from unsafe API to safe API, integrated into Python