Scylla: Translating an Applicative Subset of C to Safe Rust

AYMERIC FROMHERZ, Inria, France
JONATHAN PROTZENKO, Google, USA

OOPSLA ‘26

Benefits of Rust

- Rust combines high performance and memory safety by design
- Makes sense for a new project (clean-slate code)
- Why translate battle-tested (thoroughly debugged) C code into Rust?

C to Rust

- To assist transition, automatic translation from C to Rust

- However, these tools target unsafe Rust (which allows the use of C-like idioms)

- Problem
  - Automated C to Rust translation vs Actual memory-safety guarantees

C-to-Rust Approaches

- C2Rust descendants
  - C2Rust is a C99→Rust translator
  - Automate rewriting process that gradually removes unsafe

- LLM-based translation
  - Possible-faulty translation might be acceptable
  - What if safety of the produced rust code is paramount?

- Refactor C (This Paper)
  - Rewrite the C to use abstractions that encode Rust's borrow checking discipline

Key Approach

- Refactor the source, not the output

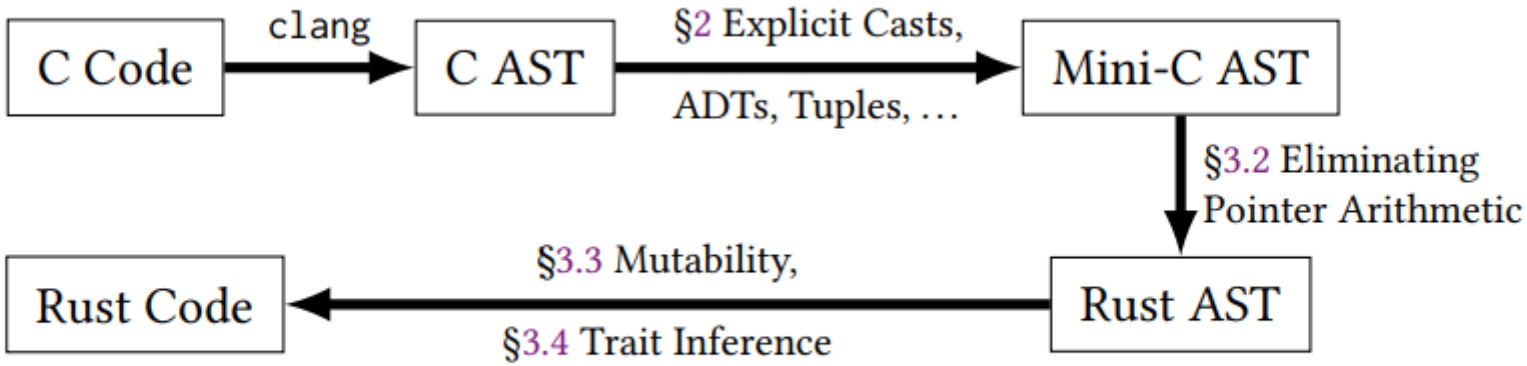


Fig. 1. Overview of the Scylla Translation

Mini-C

- Mini-C has “no-surprises” semantics
  - Integers have fixed width
  - Integer promotion or conversions are represented explicitly using casts

$t ::=$	uint8_t, bool, ...	integers, booleans...
	unit	empty type
	size_t	pointer size
	$t*$	pointer type
	$t(\vec{t})$	function type
	struct s	struct type
	$(\vec{t})$	tuple type
	variant v	variant type
$d ::=$	struct $s\,\{t_i\, f_i; \ldots t_j\, f_j[n_j];\}$	struct definition
	variant $v\, C_i : \vec{t_i}$	variant definition
$e ::=$	return e, if (e) then e else e,	
	while (e) e, continue, break	control-flow
	$x = e$	assignment
	let $t\ x(= e)?$ in e	variable declaration
	let $t\ x[n](= \{\vec{e}\})?$ in e	array declaration
	match e with $C_i\ \vec{v}_i \rightarrow e_i$	pattern-matching
	x	variable
	$e \bowtie e$ with $\bowtie = +, -, \times, \ldots$	binary operators
	$e[e]$	array indexing
	$e.f$	field selection
	$e.i$	tuple field selection
	$f(\vec{e})$	function call
	$*e$	dereference
	malloc(t, e)	typed heap allocation
	$\&e$	address-taking
	$C\,\vec{e}$	variant value
	$(t)e$	cast
	$(e_1, \ldots, e_n)$	tuple ($n > 1$)
	$()$	unit value
	$n : t$	typed integer constant

Fig. 2. Grammar of Mini-C types, statements and expressions. We use $e_1; e_2$ as syntactic sugar for let _ =

Example of Mini-C Code

```c
#define CaseU8 0
#define CaseU16 1

typedef struct
__attribute__((annotate("scylla_adt")))
cases_s {
  uint8_t tag;
  union {
    uint8_t case_uint8;
    uint16_t case_uint16;
  };
} cases;

void f(cases s) {
  if (s.tag == CaseU8) {
    uint8_t x = s.case_uint8;
  }
}
```

```
variant cases
  | case_uint8 (v: uint8_t)
  | case_uint16 (v: uint16_t)

unit f(cases s) {
  match s with
  | case_uint8 v -> let uint8_t x = v in ()
  | _ -> ()
}
```

Fig. 4. Example of tagged union translation from C (left) to Mini-C (right)

Pointer Arithmetic

The paper covers multiple nuances of translating C to Rust. We do not cover everything instead on one aspect where they introduce a challenge and propose their novel solution.

Handling Pointer Arithmetic

- C Programs rarely perform accesses and updates via a single base pointer.

```c
uint8_t abcd[32] = { 0 };

uint8_t *a = abcd + 0;
uint8_t *c = abcd + 16;
uint8_t *b = abcd + 8;
uint8_t *d = abcd + 24;
```

```rust
let mut abcd = [0u8; 32];
let abcd: &mut [u8] = &mut abcd[..];
let (a_l, a_r) = abcd.split_at_mut(0);
let (c_l, c_r) = a_r.split_at_mut(16);
let (b_l, b_r) = c_l.split_at_mut(8);
let (d_l, d_r) = c_r.split_at_mut(8);
let (a, b, c, d) = (b_l, b_r, d_l, d_r)
```

Fig. 7. Example of pointer arithmetic translation as Rust splits. For readability, we destructure tuples instead of using tuple variables and accessors as described in Figure 8.

Problems with Pointer Arithmetic

- No Length information (neither at run-time nor in the type system)
- Cannot assume that pointer arithmetic occurs in left-to-right order

```
1  uint8_t abcd[32] = { 0 };
2
3  uint8_t *a = abcd + 0;
4  uint8_t *c = abcd + 16;
5  uint8_t *b = abcd + 8;
6  uint8_t *d = abcd + 24;
7
```

```
1  let mut abcd = [0u8; 32];
2  let abcd: &mut [u8] = &mut abcd[..];
3  let (a_l, a_r) = abcd.split_at_mut(0);
4  let (c_l, c_r) = a_r.split_at_mut(16);
5  let (b_l, b_r) = c_l.split_at_mut(8);
6  let (d_l, d_r) = c_r.split_at_mut(8);
7  let (a, b, c, d) = (b_l, b_r, d_l, d_r)
```

Fig. 7. Example of pointer arithmetic translation as Rust splits. For readability, we destructure tuples instead of using tuple variables and accessors as described in Figure 8.

Split_at_mut

- Rust does not allow arbitrary pointer arithmetic

```
pub const fn split_at_mut(&mut self, mid: usize) -> (&mut [T], &mut [T])
```

1.0.0 (const: 1.83.0) · Source

Divides one mutable slice into two at an index.

The first will contain all indices from `[0, mid)` (excluding the index `mid` itself) and the second will contain all indices from `[mid, len)` (excluding the index `len` itself).

Panics

Panics if `mid > len`. For a non-panicking alternative see `split_at_mut_checked`.

Examples

```
let mut v = [1, 0, 3, 0, 5, 6];
let (left, right) = v.split_at_mut(2);
assert_eq!(left, [1, 0]);
assert_eq!(right, [3, 0, 5, 6]);
left[1] = 2;
right[1] = 4;
assert_eq!(v, [1, 2, 3, 4, 5, 6]);
```

Difficulties

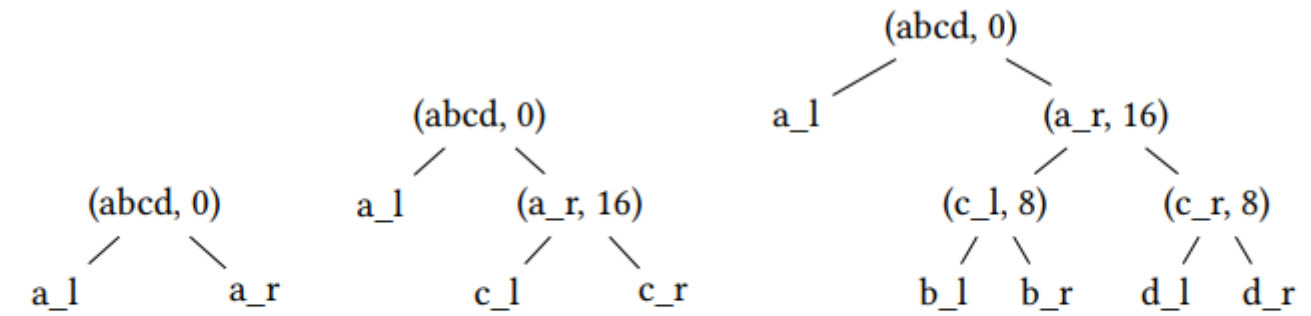
- C doesn’t provide length
  - We assume that chunks are not intended to be overlapping
- Translation needs to be predictable and understandable
  - translation failure should be recognizable

Split Trees

```
1  uint8_t abcd[32] = { 0 };
2
3  uint8_t *a = abcd + 0;
4  uint8_t *c = abcd + 16;
5  uint8_t *b = abcd + 8;
6  uint8_t *d = abcd + 24;
7
```

```
1  let mut abcd = [0u8; 32];
2  let abcd: &mut [u8] = &mut abcd[..];
3  let (a_l, a_r) = abcd.split_at_mut(0);
4  let (c_l, c_r) = a_r.split_at_mut(16);
5  let (b_l, b_r) = c_l.split_at_mut(8);
6  let (d_l, d_r) = c_r.split_at_mut(8);
7  let (a, b, c, d) = (b_l, b_r, d_l, d_r)
```

Fig. 7. Example of pointer arithmetic translation as Rust splits. For readability, we destructure tuples instead of using tuple variables and accessors as described in Figure 8.



(a) After translating a (b) After translating c (c) After translating b and d

Fig. 9. Successive split trees during C translation. Internal nodes of the form (x, i) have been subjected to a split at Rust index i and are therefore borrowed at this program point. Leaf nodes are available.

What if offsets aren’t constant?

- Symbolic Solver
  - E.g., n+8 > n
  - If solver fails, implementation emits a warning along with corresponding location (e.g., 2*n → n+n)
  - If all fails, assume that offsets that occur along the control-flow are monotonically increasing

Performance

Table 1. Performance comparison. Results are normalized taking the original C code as a baseline.

Benchmark	Original C	Rewritten C	Rust
SymCrypt SHA3-256	1	1.00	1.00
SymCrypt SHA3-384	1	1.00	1.01
SymCrypt SHA3-512	1	1.00	1.00
Frodo640KEM KeyGen	1	.98	.97
Frodo640KEM Encaps	1	.99	.98
Frodo640KEM Decaps	1	.99	.99
HACL* Curve25519	1	1.02	.99
HACL* SHA2-256	1	.99	1.05
HACL* ChachaPoly-Enc	1	1.00	.99
HACL* ECDH-P256	1	.98	.77
bzip2-compress	1	.99	1.08
EverParse CBOR 2200	1	1.00	.87
EverParse CBOR 1000	1	1.00	1.21
EverParse CBOR 250	1	1.00	1.22
EverParse CBOR 2200 (inline)	1	1.00	.65
EverParse CBOR 1000 (inline)	1	1.00	.91
EverParse CBOR 250 (inline)	1	1.00	.91

Comparison with C2Rust

```
1  void chacha20_encrypt_block(uint32_t *ctx, uint8_t *out, uint32_t incr, uint8_t *text) {
2    uint32_t k[16U] = { 0U };
3    chacha20_core(k, ctx, incr);
4    uint32_t bl[16U] = { 0U };
5    for (int i = 0; i < 16; i++) {
6      uint8_t *bj = text + i * 4U;
7      uint32_t u = load32_le(bj);
8      uint32_t *os = bl;
9      os[i] = u;
10 }
```

Comparison with C2Rust

```rust
pub fn chacha20_encrypt_block(ctx: &[u32], out: &mut [u8], incr: u32, text: &[u8]) {
  let mut k: [u32; 16] = [0u32; 16usize];
  chacha20_core(&mut k, ctx, incr);
  let mut bl: [u32; 16] = [0u32; 16usize];
  for i in 0u32..16u32 {
    let bj: (&[u8], &[u8]) = text.split_at(i.wrapping_mul(4u32) as usize);
    let u: u32 = load32_le(bj.1);
    let os: (&mut [u32], &mut [u32]) = bl.split_at_mut(0usize);
    os.1[i as usize] = u }
}
```

```rust
unsafe extern "C" fn chacha20_encrypt_block(
    mut ctx: *mut uint32_t, mut out: *mut uint8_t, mut incr: uint32_t, mut text: *mut uint8_t,
) {
  let mut k: [uint32_t; 16] = [0 as libc::c_uint, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,];
  chacha20_core(k.as_mut_ptr(), ctx, incr);
  let mut bl: [uint32_t; 16] = [0 as libc::c_uint, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,];
  let mut i: uint32_t = 0 as libc::c_uint;
  while i < 16 as libc::c_uint {
    let mut bj: *mut uint8_t = text.offset(i.wrapping_mul(4 as libc::c_uint) as isize);
    let mut u: uint32_t = load32_le(bj);
    let mut os: *mut uint32_t = bl.as_mut_ptr();
    *os.offset(i as isize) = u;
    i = (i as libc::c_uint).wrapping_add(1 as libc::c_uint) as uint32_t as uint32_t; }
}
```

Fig. 11. Translation of encrypt_block with Scylla (top) and c2rust (bottom)

Undefined Behavior

- Scylla identified Undefined Behavior on bzip2 and Microsoft's implementation of FrodoKEM

- Bzip2
  - Two sources (alignment issue and strict aliasing violation)

- FrodoKEM
  - Violates strict aliasing
  - Cryptographic code oftentimes converting between uint16_t* and uint32_t*

Limitations of Scylla

- Gotos and unstructured control flow
- Leverage object representation through integer-to-pointer casts
- Pointer tricks
- Bitfields
- Untagged unions

Closing Remarks

- Is “unsafe” really that bad?
  - Unsafe can be translated to safe using wrappers but readability and maintainability should also be important for security critical code

- Clang frontend
  - Scylla makes use of clang’s frontend. Many C codebases rely on GCC-specific extensions