The C Language as a Translation Source: Semantics, Undefined Behavior, and the libyaml Idiom Set

A Reference for the Ferrous Bridge C-to-Rust Pipeline

Matthew Long
The YonedaAI Collaboration, YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com · https://yonedaai.com

16 April 2026

Abstract

The Ferrous Bridge programme translates security-critical C libraries into safe, idiomatic Rust. The first concrete target is `libyaml` (approximately 9,300 lines of C across ten source files), whose output crate is named `safe-libyaml`. To translate C faithfully one must read C as a *specification language for invariants* rather than as a procedural recipe: the ISO C18 abstract machine leaves vast latitude (sequence points, indeterminate evaluation order, undefined behaviour, type-based aliasing assumptions) which C compilers exploit aggressively. This paper provides the Rust systems engineer who will perform or review such translations with a unified semantic reference. We give a small-step operational semantics for a translation-relevant fragment of C, organise undefined behaviour into a taxonomy aligned with the Rust type-system features that statically forbid each class, catalogue the recurring C idioms found in `libyaml` (the three-pointer `yaml_string_t`, the `STACK_*`/`QUEUE_*`/`STRING_*` macro families, the integer-as-bool return convention, the tagged-union event type, the custom allocator hooks, scanner-into-parser buffer aliasing, nullable optional pointers, and the ambient parser-resident error field), and for each idiom we state a *translation contract*: the Rust counterpart to which it must map and the proof obligation that must be discharged for the translation to be faithful. We close with an appendix giving a development plan of twenty-nine prototype-agent tasks for the analysis-phase of the Ferrous Bridge pipeline that produces the C-Analysis Bundle (CAB).

Contents

1 Introduction

The Ferrous Bridge pipeline is an AI-assisted toolchain whose stated objective is to convert security-critical C codebases into *safe* Rust crates — crates that compile under `rustc` with `#![deny(unsafe_code)]` outside narrowly scoped FFI shims, that pass under Miri's stacked-borrows interpreter, and that survive at least 10^6 differential fuzzing iterations against the original C reference. The first concrete proving ground is `libyaml`: ninety million transitive downloads of `serde_yaml` and its forks currently flow through `unsafe-libyaml` [1], the `c2rust` [2] mechanical transpilation. Our goal is to publish `safe-libyaml` as a drop-in safe replacement.

To produce safe Rust from C one cannot translate syntactically. Every C pointer arithmetic expression, every `int` return code, every `void *` callback context conceals a *contract* that the original author held only in their head. The contract specifies who owns the pointee, when it may be aliased, when it may be freed, what range of indices is in-bounds, and which return value transitions denote success. C's standard does not require these contracts to be written down; the compiler does not check them; the cost is that the contracts are routinely violated and the violations become CVEs. This paper exists to make those contracts explicit so that an analysis agent (or a careful human) can re-state them in the Rust type system.

The paper is structured as follows. Section 2 gives a small-step operational semantics for the fragment of C needed to talk about translation: lvalues, rvalues, sequence points, evaluation order, indeterminate values. Section 3 organises undefined behaviour into seven classes and gives, for each, the Rust feature that statically forbids the class. Section 4 catalogues the `libyaml` idiom set: every concrete pattern that an analysis agent will find in the source. Section 5 formalises a translation contract for each idiom: an inference rule that says "if this C idiom is observed and these side conditions are discharged, then the corresponding Rust construct is a refinement." Section 8 surveys related work — KCC, CompCert, c2rust, the recent agentic transpilation literature. Section 10 ties the contributions back to the companion Rust and `c-rust-transpiling` papers. Section A is a development plan with twenty-nine checkbox tasks for the analysis-phase prototype agents that produce the CAB.

1.1 Pipeline preliminaries: CAB, OSG, and the agent stages

Although this paper stands on its own as a reference on C semantics for translation, several artifact names recur and are best fixed up front. The Ferrous Bridge pipeline has five stages, of which the first two are relevant here.

Stage 1 (C Frontend Analysis) consumes C source files and produces the *C-Analysis Bundle (CAB)*: a Protocol-Buffers-serialised package containing per-translation-unit Clang ASTs, LLVM IR, control-flow graphs, the call graph, an Andersen-style points-to graph, use-def chains, and dynamic-trace summaries from sanitiser runs. The CAB is the typed input to Stage 2.

Stage 2 (Ownership Semantic Graph) consumes the CAB and emits the *Ownership Semantic Graph (OSG)*: a typed intermediate representation in which every C pointer parameter is annotated with one of five ownership classes (Own, $\mathsf{Bor}^{\mathrm{shr}}$, $\mathsf{Bor}^{\mathrm{mut}}$, Shar, Raw), a confidence score in $[0, 1]$, and a Rust idiom hint (`Vec`-from-malloc-array, `String`-from-

char-star, `Option`-from-nullable-pointer, etc.). The downstream translation engine (Stage 3) routes each function to either a fine-tuned model or to Claude based on the confidences.

The translation contracts of Section 5 are the rule-set against which the OSG is built: an OSG entry is precisely the discharged or undischarged side condition of one of the contracts. The development plan of Section A is the build plan for the agents that produce the CAB and lift it to the OSG.

1.2 Contribution

The specific contributions of this paper are:

1. A small-step operational semantics, restricted to the translation-relevant C fragment, in which sequence points are first-class and indeterminate values are explicit (Section 2).

2. A seven-class undefined-behaviour taxonomy, with each class mapped to the precise Rust type-system feature that statically rules it out (Section 3).

3. A complete catalogue of the `libyaml` idiom set, lifted from five core source files in `libyaml`—`yaml_private.h`, `api.c`, `scanner.c`, `parser.c`, and `emitter.c`—with annotated C excerpts (Section 4).

4. Eleven *translation contracts*, one per `libyaml` idiom, each a typed refinement statement together with its proof obligation (Section 5).

5. A twenty-nine task development plan for the analysis-phase agents that build the C-Analysis Bundle (Section A).

1.3 Notation

We write $\langle e,\ \sigma\rangle$ for a C machine configuration consisting of an expression e and a store $\sigma : \ell \rightharpoonup v$. The relation $\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle$ is small-step reduction. The relation $e_1 \prec_{\mathrm{sb}} e_2$ is *sequenced before* in the sense of ISO C18 §5.1.2.3p3. We write $e \sqsubseteq_T e'$ for *semantic refinement at type* T: every behaviour of the refining program is a permitted behaviour of the refined program. Translation is written as a function $\mathcal{T} : \mathcal{C} \to \mathcal{R}$, where $\mathcal{C}$ is a fragment of well-defined C programs and $\mathcal{R}$ is safe Rust.

2 The ISO C18 Abstract Machine

2.1 Why we need a formal semantics

Idiomatic C translation work in industry usually proceeds informally: the engineer reads the C, mentally simulates execution, and writes Rust. Mental simulation is unsound for C for three reasons. First, the standard licenses many evaluation orders and the ones that occur in production are compiler- and optimisation-flag dependent. Second, the standard *licenses arbitrary behaviour* on undefined-behaviour-tripping programs, and modern compilers exploit this license for optimisation; the actually-observed behaviour of a C program is therefore not

a sound reference for translation. Third, C pointers carry implicit metadata — provenance, alignment, lifetime — that has no syntactic representation in the source. We need a formal model where these are explicit.

We do not give a full semantics; the canonical reference is the KCC executable semantics of Ellison and Roşu [3]. We give the minimum needed to talk about translation contracts.

2.2 Syntax of the translation fragment

$$
\begin{aligned}
\tau &::= \texttt{int} \mid \texttt{char} \mid \tau * \mid \texttt{struct}\ s \mid \texttt{union}\ u \mid \texttt{void} \\
e &::= n \mid x \mid \&e \mid *e \mid e_1\ op\ e_2 \mid e_1 = e_2 \mid e_1[e_2] \mid e.f \mid e \to f \\
&\mid\ (\tau)e \mid e_1(e_2, \ldots) \mid \texttt{sizeof}(\tau) \mid e_1, e_2 \\
\mathit{stmt} &::= e; \mid \texttt{return}\ e; \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ (e)\ s \mid \{\mathit{stmt}^*\}
\end{aligned}
$$

The fragment omits goto, switch, and bit-fields; `libyaml` uses these only sparingly and they pose no novel translation difficulty.

2.3 Stores, locations, values

Definition 2.1 (Store)**.** A *store* is a partial map $\sigma : \ell \times \mathbb{N} \rightharpoonup v \cup \{\mathsf{Indet}\}$, indexed by a base location ℓ and a byte offset. The byte at offset i of object ℓ may be a determinate value or the distinguished tag Indet ("indeterminate"), modelling fresh `malloc`-returned memory or uninitialised stack slots.

Definition 2.2 (Provenance-bearing pointer value)**.** A pointer value is a triple (ℓ, i, p) where $\ell \in \ell$ is the base object, $i \in \mathbb{Z}$ is the offset in bytes from the base, and $p \in \{\mathsf{Live}, \mathsf{Dead}\}$ is the *provenance* of the object (whether the object is currently allocated). A pointer with $p = \mathsf{Dead}$ traps any access.

2.4 Reading indeterminate bytes is undefined

Theorem 2.3 (ISO C18 §6.2.6.1p5, simplified)**.** *If* $\langle e,\ \sigma\rangle \longrightarrow^* \langle *(\ell, i, \mathsf{Live}),\ \sigma'\rangle$ *and there exists a byte offset* $j \in [i, i + \mathit{sizeof}(\tau))$ *with* $\sigma'(\ell, j) = \mathsf{Indet}$*, and* τ *is not* `char`*,* `signed char`*, or* `unsigned char`*, then the program exhibits undefined behaviour.*

The translation consequence: any C struct allocated by `malloc` (rather than `calloc`) and read field-wise before being initialised is UB. Rust forbids this statically: every field of a `struct` value must be initialised at construction.

2.5 Sequence points and the sequenced-before relation

Definition 2.4 (Sequenced-before)**.** The relation $\prec_{\mathrm{sb}}$ is the smallest partial order over expression evaluations satisfying the rules of ISO C18 §6.5: the left operand of `&&`, `||`, `,`, and `?:` is sequenced before the right; the function-call operand expressions are sequenced before the call; assignment is sequenced after both operands. Operands of arithmetic and most other binary operators are *not* ordered.

Theorem 2.5 (Unsequenced modification is UB, ISO C18 §6.5p2)**.** *If two side effects on the same scalar object are* unsequenced *(neither* $\prec_{\mathrm{sb}}$ *the other), the program exhibits undefined behaviour.*

The classic example `i = i++` is undefined for this reason. Rust statically forbids this: the borrow checker prevents two simultaneous `&mut` references to the same place.

2.6 Small-step rules (selected)

We give the rules that matter for the translation contracts in Section 5. Two simplifications relative to a fully-formal C semantics deserve note. First, we conflate the lvalue/rvalue distinction in the rules below: Var produces the stored value directly, while Assign re-evaluates the LHS as a location. A complete semantics would have separate evaluation judgements for lvalues and rvalues with an explicit lvalue-to-rvalue conversion (ISO C18 §6.3.2.1); the conflation does not affect the translation contracts that follow. Second, we omit type promotion and the integer rank rules (§6.3.1) for arithmetic expressions; these are orthogonal to the UB classes that drive translation.

$$\textsc{Var}\ \frac{\sigma(\Gamma(x)) = v}{\langle x,\ \sigma\rangle \longrightarrow \langle v,\ \sigma\rangle} \qquad \textsc{AddrOf}\ \langle \&x,\ \sigma\rangle \longrightarrow \langle (\Gamma(x), 0, \mathsf{Live}),\ \sigma\rangle$$

$$\textsc{Deref-Live}\ \frac{p = (\ell, i, \mathsf{Live}) \qquad \sigma(\ell, i) = v \neq \mathsf{Indet}}{\langle *p,\ \sigma\rangle \longrightarrow \langle v,\ \sigma\rangle} \qquad \textsc{Deref-Dead}\ \frac{p = (\_, \_, \mathsf{Dead})}{\langle *p,\ \sigma\rangle \longrightarrow \bot_{ub}}$$

$$\textsc{Assign}\ \frac{\langle e_2,\ \sigma\rangle \longrightarrow^* \langle v,\ \sigma_1\rangle \qquad \langle e_1,\ \sigma_1\rangle \longrightarrow^* \langle (\ell, i, \mathsf{Live}),\ \sigma_2\rangle}{\langle e_1 = e_2,\ \sigma\rangle \longrightarrow \langle v,\ \sigma_2[(\ell, i) \mapsto v]\rangle}$$

$$\textsc{PtrAdd-InBounds}\ \frac{p = (\ell, i, \mathsf{Live}) \qquad 0 \le i + n \le \mathit{sizeof}(\ell)}{\langle p + n,\ \sigma\rangle \longrightarrow \langle (\ell, i + n, \mathsf{Live}),\ \sigma\rangle} \qquad \textsc{PtrAdd-OOB}\ \frac{p = (\ell, i, \mathsf{Live}) \qquad \neg(0 \le i + n \le \mathit{sizeof}(\ell))}{\langle p + n,\ \sigma\rangle \longrightarrow \bot_{ub}}$$

$$\textsc{Deref-OnePastEnd}\ \frac{p = (\ell, \mathit{sizeof}(\ell), \mathsf{Live})}{\langle *p,\ \sigma\rangle \longrightarrow \bot_{ub}}$$

The key features of these rules follow ISO C18 §6.5.6p8 precisely. Pointer formation is licensed throughout the closed interval $[0, \mathit{sizeof}(\ell)]$ — one byte past the end of the allocation is a *valid* pointer value that may be formed and compared, but not dereferenced. Rule PtrAdd-InBounds captures this: the upper bound is the inclusive $\le$. Rule Deref-OnePastEnd singles out the dereference of a one-past-the-end pointer as the UB case; merely holding such a pointer is fine. Rule PtrAdd-OOB traps when arithmetic would land strictly outside the closed interval. Dereferencing a `Dead` pointer traps to $\bot_{ub}$; reading an Indet byte (rule omitted; same shape) traps to $\bot_{ub}$. These four classes capture

buffer-overflow, use-after-free, and uninitialised-read CVEs that account for the majority of exploitable bugs in `libyaml`, `libexpat`, `libpng`, and `libxml2`.

2.7 Refinement to Rust

Definition 2.6 (Observable trace)**.** For a program P on input x, the *observable trace* $\tau(P, x)$ is the (possibly infinite) sequence of externally visible events: I/O calls (with arguments), the final return value (if the program terminates), and one of the marker events $\bot_{ub}$ (UB encountered) or $\bot_{err}$ (a defined error reported through the language's error mechanism). Internal memory operations are not observable.

Definition 2.7 (Translation refinement)**.** A translation $\mathcal{T}(P) = Q$ is a *refinement at observation type* O if for every input x:

1. if $\bot_{ub} \notin \tau(P, x)$, then $\pi_O(\tau(Q, x)) = \pi_O(\tau(P, x))$; and
2. if $\bot_{ub} \in \tau(P, x)$, then $\tau(Q, x)$ may be any trace at all, including $\bot_{err}$ (returning a defined `Error`) or any well-defined I/O sequence followed by termination.

The projection π_O abstracts away semantically-irrelevant differences such as the choice of memory allocator (`malloc` vs. `Vec::push`) or the internal layout of an `enum`.

This definition follows the standard refinement notion of verified compilation [6]: the refining program may have *strictly fewer* behaviours than the refined program on every input. The asymmetry on UB inputs is intentional. We refuse to require that Q reproduce P's behaviour on inputs where P is undefined: such reproduction would preserve compiler-exploitable assumptions and would amount to translating bugs faithfully. We adopt this orientation throughout the paper: the goal of the pipeline is to maximise the number of implicit C invariants that are promoted to compile-time enforcement [17], while remaining strictly conservative on UB inputs.

3 Undefined Behaviour Taxonomy

The ISO C18 standard enumerates more than two hundred undefined behaviours. For translation purposes we collapse them into seven classes, chosen so that each is statically ruled out by a single Rust feature.

3.1 Class UB1: Spatial memory safety

Examples: buffer overread/overwrite, OOB pointer arithmetic, stack overflow.
C source: `*(p + i)` when $i \geq$ *sizeof*$(*p)/$`sizeof`$(*p)$; `strcpy` into a buffer too small.
Rust feature: slice indexing `s[i]` which panics on OOB; the borrow-checked `&[T]` and `&mut [T]` carry length at the type level.
Defining CVE class for `libyaml`: the historical heap overflows in scalar scanning. The `yaml_string_t` triple {`start`, `end`, `pointer`} has no compiler-enforceable guarantee that `pointer` stays within [`start`, `end`]; the equivalent `Vec<u8>` carries length statically, and indexing is checked.

3.2 Class UB2: Temporal memory safety

Examples: use-after-free, double-free, dangling reference.
C source: `free(p); use(p);` or `free(p); free(p);`.
Rust feature: ownership: a value has exactly one owner; `drop` runs at scope end; the borrow checker prevents reference outliving the referent.
Translation note: the `STACK_DEL` and `STRING_DEL` macros free the buffer and zero the pointers — a manual workaround for use-after-free that becomes automatic in Rust.

3.3 Class UB3: Type-based aliasing (TBAA)

Examples: accessing the same storage through pointers of unrelated effective types; classic `int *` and `float *` aliasing.
C source: `*(int *)&f` where `f` is `float`.
Rust feature: `&T` and `&mut T` encode aliasing rules in the type system; `transmute` requires `unsafe`; LLVM's `noalias` metadata is licensed soundly because the borrow checker has discharged it.
Translation note: `libyaml`'s scanner reads ahead into the parser's buffer through pointers that alias the same storage; the translation contract (Section 5) replaces this with explicit slice borrows.

3.4 Class UB4: Signed integer overflow

Examples: `INT_MAX + 1`; left shift past the type width; division of `INT_MIN` by -1.
C source: `int n = a + b;` when $a + b >$ `INT_MAX`.
Rust feature: `i32::checked_add`, `wrapping_add`, `overflowing_add`; in debug builds, plain `+` panics on overflow; in release, two's-complement wrap.
Translation note: libyaml computes `capacity * 2` during `STACK_EXTEND` which on 32-bit systems can overflow `size_t` on a sufficiently large input. The Rust translation must use `checked_mul` and report an error.

3.5 Class UB5: Unsequenced modification

Examples: `i = i++;` `a[i] = i++;` passing `f(i, i++)`.
C source: two side effects on the same scalar with no sequence point between.
Rust feature: aliasing-XOR-mutability: at most one `&mut` live at a time, never simultaneous with `&`.
Translation note: `libyaml` uses macro-expanded post-increments inside expressions; the macro expansions must be lifted to statements during translation.

3.6 Class UB6: Indeterminate values and uninitialised reads

Examples: reading from `malloc`-returned memory before writing.
C source: `int *p = malloc(sizeof *p); printf("%d", *p);`
Rust feature: all fields of a `struct` value must be initialised at construction; invoking

`MaybeUninit<T>::assume_init` requires `unsafe`.
Translation note: In `libyaml`, `yaml_parser_initialize` zeroes the parser with `memset`. The Rust translation instead provides a `Default` implementation or an explicit constructor that initialises every field.

3.7 Class UB7: Concurrent data races

Examples: unsynchronised read-write or write-write to the same memory from two threads.
C source: `*p++` from one thread, `*p++` from another, no atomics, no mutex.
Rust feature: `Send` and `Sync` marker traits; sharing across threads requires `Arc<Mutex<T>>` or `Arc<RwLock<T>>`; types that are not `Sync` (such as `Cell<T>`) cannot be shared.
Translation note: libyaml is single-threaded; this class is mostly defensive.

3.8 Tabular summary

Table 1 summarises the seven classes and their Rust preventions.

Class	C source	Rust prevention
UB1 spatial	`*(p + n)` OOB	checked indexing on `&[T]`
UB2 temporal	`free(p); *p`	ownership, `drop`
UB3 TBAA	punning casts	`&T` / `&mut T` typing
UB4 signed overflow	`INT_MAX + 1`	`checked_add`, debug panic
UB5 unsequenced	`i = i++`	&-XOR-`&mut` discipline
UB6 indeterminate	`malloc` then read	full-field initialisation
UB7 race	unsync threaded access	`Send`, `Sync`, `Mutex`

Table 1: Mapping of C undefined-behaviour classes to the Rust feature that statically forbids each.

4 C Idioms Relevant to libyaml and Similar Libraries

This section presents every recurring C pattern that an analysis agent will find in `libyaml`, plus one closely-related pattern (`setjmp/longjmp`, idiom I11) that does not occur in `libyaml` itself but does occur in adjacent translation targets such as `libpng`; we include it because it is the limit case against which our framework's handling must be measured. The catalogue is exhaustive for `libyaml`: every CAB (Section A) entry for a `libyaml` function classifies into one or more of idioms I1–I10.

4.1 Idiom I1: The yaml_string_t triple

```c
typedef struct {
    yaml_char_t *start;   /* beginning of allocated buffer */
    yaml_char_t *end;     /* one past end of allocated buffer */
    yaml_char_t *pointer; /* current cursor position */
```

```
} yaml_string_t;
```

Listing 1: From `yaml_private.h`

The triple manually encodes *capacity* = `end` − `start`, *position* = `pointer` − `start`, and *remaining* = `end` − `pointer`. The invariant `start` ≤ `pointer` ≤ `end` is the programmer's responsibility; nothing in the type system enforces it. Every read through `*pointer` or `*(pointer + k)` risks UB1.

4.2 Idiom I2: The STRING_*, STACK_*, QUEUE_* macro families

```
#define STRING_INIT(context, string, size)                              \
    (((string).start = yaml_malloc(size)) ?                             \
        ((string).pointer = (string).start,                             \
         (string).end = (string).start + (size),                        \
         memset((string).start, 0, (size)),                             \
         1) :                                                           \
        ((context)->error = YAML_MEMORY_ERROR, 0))

#define STRING_EXTEND(context, string)                                  \
    ((((string).pointer + 5 < (string).end)                             \
        || yaml_string_extend(&(string).start,                          \
            &(string).pointer, &(string).end)) ?                        \
         1 :                                                            \
         ((context)->error = YAML_MEMORY_ERROR, 0))

#define STACK_INIT(context, stack, size)        /* analogous */
#define STACK_PUSH(context, stack, value)       /* analogous */
#define STACK_POP(context, stack)               /* analogous */
#define QUEUE_INIT(context, queue, size)        /* ring buffer */
#define ENQUEUE(context, queue, value)          /* analogous */
#define DEQUEUE(context, queue)                 /* analogous */
```

The macro families fuse pointer arithmetic, error handling (set the parser's `error` field), and dynamic allocation. `INITIAL_STACK_SIZE = 16`; `STACK_PUSH` doubles capacity via `realloc` when full. The family expands inline at every call site, which makes the code visually complex and trivially defeats most cross-function analysis.

4.3 Idiom I3: integer-as-bool return convention

```
int yaml_parser_parse(yaml_parser_t *parser, yaml_event_t *event);
/* returns 1 on success, 0 on failure; the error is in parser->error */
```

Every `libyaml` parser/emitter function returns `int` where 1 means success and 0 means failure. The error itself is stored in the parser/emitter struct as the field `error`, with the human-readable message in `problem` and the location in `problem_mark`. Callers who forget the `if (!yaml_parser_parse(&p, &e)) goto error;` pattern silently process garbage.

4.4 Idiom I4: output-parameter convention

The return type carries the success bit; the actual produced value is delivered via an out-parameter. The caller pre-allocates an `yaml_event_t` on the stack and passes its address. The pattern arose because C lacks tuple returns and because `yaml_event_t` is large (≈ 100 bytes) so returning by value would copy.

4.5 Idiom I5: the yaml_event_t tagged union

```
typedef struct yaml_event_s {
    yaml_event_type_t type;
    union {
        struct { yaml_encoding_t encoding; }
    stream_start;
        struct { yaml_version_directive_t *version_directive;
                 yaml_tag_directive_t    *tag_directives_start;
                 yaml_tag_directive_t    *tag_directives_end;
                 int                      implicit; }
    document_start;
        struct { int implicit; }
    document_end;
        struct { yaml_char_t *anchor; }                              alias;
        struct { yaml_char_t *anchor;
                 yaml_char_t *tag;
                 yaml_char_t *value;
                 size_t       length;
                 int          plain_implicit;
                 int          quoted_implicit;
                 yaml_scalar_style_t style; }                        scalar;
        struct { yaml_char_t *anchor; yaml_char_t *tag;
                 int implicit; yaml_sequence_style_t style; }
    sequence_start;
        struct { yaml_char_t *anchor; yaml_char_t *tag;
                 int implicit; yaml_mapping_style_t style; }
    mapping_start;
    } data;
    yaml_mark_t start_mark;
    yaml_mark_t end_mark;
} yaml_event_t;
```

The discriminant `type` is held alongside the `union data`. Reading a field of `data.scalar` when `type != YAML_SCALAR_EVENT` is UB6 (the bytes are technically determinate after the producer wrote them, but the type used to read is wrong, so this is also a TBAA UB3 violation). The Rust translation uses an enum, fusing the tag and the data into a single ADT.

4.6 Idiom I6: custom allocator hooks

```
typedef void *(*yaml_malloc_t)(void *data, size_t size);
typedef void *(*yaml_realloc_t)(void *data, void *ptr, size_t size);
typedef void  (*yaml_free_t)(void *data, void *ptr);
```

```
void yaml_parser_set_memory_handler(yaml_parser_t *parser,
        yaml_malloc_t  malloc,
        yaml_realloc_t realloc,
        yaml_free_t    free,
        void          *data);
```

The parser/emitter struct holds three function pointers and a `void *data` context. The indirection allows callers to use a private arena, but the `void *` loses all type information, so any miscast inside the callback is silently UB3.

4.7 Idiom I7: scanner-into-parser buffer aliasing

The scanner reads bytes into the parser's input buffer (`parser->buffer`) and exposes a cursor (`parser->buffer.pointer`) into that same buffer. Token productions take *slices* into the buffer that must remain valid until the token is consumed. The scanner can also extend the buffer (`yaml_parser_update_buffer`), which may `realloc` and invalidate any outstanding slices. There is no compiler-checked guarantee that no slice is outstanding at the time of the `realloc`; the implementation simply assumes it.

4.8 Idiom I8: nullable optional pointers

```
yaml_char_t *anchor;  /* NULL means no anchor */
yaml_char_t *tag;     /* NULL means no tag */
```

Optional fields are encoded by null pointers. A consumer who forgets the `if (anchor)` guard dereferences NULL (UB2). The Rust translation uses `Option<String>`.

4.9 Idiom I9: ambient error state in the parser

```
struct yaml_parser_s {
    yaml_error_type_t error;
    const char       *problem;
    size_t            problem_offset;
    int               problem_value;
    yaml_mark_t       problem_mark;
    /* ... thirty more fields ... */
};
```

Errors are not returned; they are *stashed* in the parser. The function returns 0 to signal that an error has been stashed. This is action-at-a-distance: the caller must look at the right field of the right struct to recover the cause.

4.10 Idiom I10: character predicate macros

```
#define IS_BLANK(string)  ((string).pointer[0] == ' '  \
                        || (string).pointer[0] == '\t')
#define IS_BREAK(string)  ((string).pointer[0] == '\r' \
                        || (string).pointer[0] == '\n' \
```

```
                          /* ... or various Unicode line breaks */)
#define CHECK(string, octet)   ((string).pointer[0] == (octet))
#define CHECK_AT(string, octet, offset) \
                               ((string).pointer[offset] == (octet))
#define FORWARD(string)        ((string).pointer++)
```

Each predicate indexes into a `yaml_string_t` at a relative offset and compares to a constant. The implicit precondition is that `pointer + offset < end`; the scanner's invariants make this true at every call site, but the precondition is unwritten.

4.11 Idiom I11: setjmp/longjmp non-local control flow

`libyaml` does not use `setjmp/longjmp`, but the closely related `libpng` does. We mention it because it is the limit case of UB-class harm: `longjmp` unwinds without running any cleanup, leaving `malloc`-ed memory leaked and locks held. Rust has no analogue; `panic::catch_unwind` is the closest, and it requires careful boundary analysis.

5 Translation Contracts

For each idiom in Section 4 we now state a *translation contract*: an inference rule whose conclusion is a typed Rust construct, whose premises are the side conditions that make the translation a refinement (Theorem 2.7). Discharging the side conditions is the work of the analysis pipeline.

5.1 Contract C1 for I1 (yaml_string_t)

String

$$\frac{x : \texttt{yaml\_string\_t} \qquad \text{capacity is the only resource invariant} \qquad \text{aliasing analysis: no live alias to } x.\texttt{start}}{\mathcal{T}(x) = \texttt{Vec} < \texttt{u8} > : \text{Rust}}$$

Proof obligation. The analysis must show that no live pointer aliases `x.start` except those that can be proven to be lexical sub-expressions of accesses through `x.pointer`. Failing this, the structure must be classified as Shar and translated to `Rc<RefCell<Vec<u8>>>`.

5.2 Contract C2 for I2 (container macros)

$$\frac{m \in \{\texttt{STACK\_PUSH}, \texttt{STACK\_POP}\} \qquad \text{stack is owned by exactly one struct field}}{\mathcal{T}(m(s,v)) \mapsto \texttt{s.push(v)} \mid \texttt{s.pop()} : \textsc{Rust}} \quad \textsc{Stack}$$

$$\frac{m \in \{\texttt{ENQUEUE}, \texttt{DEQUEUE}\}}{\mathcal{T}(m(q,v)) \mapsto \texttt{q.push\_back(v)} \mid \texttt{q.pop\_front()} : \textsc{Rust}} \quad \textsc{Queue}$$

$$\frac{m \in \{\texttt{STRING\_EXTEND}, \texttt{STRING\_JOIN}\}}{\mathcal{T}(m(s,t)) \mapsto \texttt{s.extend\_from\_slice(t)} \mid \texttt{s.reserve(n)} : \textsc{Rust}} \quad \textsc{String-Macro}$$

Proof obligation. The container is owned: malloc and free occur in the same scope (or an unambiguous owner-droppee pair).

5.3 Contract C3 for I3 (integer-as-bool)

$$\frac{f : \texttt{int}(*)(\ldots) \qquad \forall \text{ call site} : \texttt{0} = \texttt{error}, \texttt{1} = \texttt{ok}}{\mathcal{T}(f) : \texttt{fn}(\ldots) \to \texttt{Result} < \texttt{T}, \texttt{Error} >} \quad \textsc{IntBool}$$

Proof obligation. Every documented call site checks the return; no callee uses a return value other than 0 or 1.

5.4 Contract C4 for I4 (output parameter)

$$\frac{f(\texttt{ctx}, \texttt{out}) : \texttt{int} \qquad \texttt{out} \text{ is written exactly when } f \text{ returns } 1}{\mathcal{T}(f) : \texttt{fn}(\&\texttt{mut Ctx}) \to \texttt{Result} < \texttt{Out}, \texttt{Error} >} \quad \textsc{OutParam}$$

Proof obligation. The output parameter is written on the success path and unmodified on the failure path. The static analyser checks this via use-def chains.

5.5 Contract C5 for I5 (tagged union)

$$\frac{\begin{array}{c} t : \texttt{enum T} \{\texttt{V}_1, \ldots \texttt{V}_\texttt{n}\} \\ U : \texttt{union} \{V_1\ data_1; \ldots; V_n\ data_n;\} \qquad \forall \text{ access } U.data_i \text{ guarded by } t = V_i \end{array}}{\mathcal{T}(\texttt{struct} \{t; U;\}) = \texttt{enum T} \{\texttt{V}_1(data_1), \ldots \texttt{V}_\texttt{n}(data_\texttt{n})\}} \quad \textsc{Tagged}$$

Proof obligation. For every read of $\texttt{u}.data_i$ the analysis must witness a dominator condition $\texttt{t == V}_i$. This is straightforward control-flow analysis when the tag is checked in a `switch`; difficult when checked in disjoint `if` chains.

5.6 Contract C6 for I6 (allocator hooks)

ALLOC

$$\frac{f_m, f_r, f_f \text{ are passed to } \texttt{yaml\_parser\_set\_memory\_handler} \quad f_m, f_r, f_f \text{ are pure wrappers around } \texttt{malloc}, \texttt{realloc}, \texttt{free}}{\mathcal{T}(\texttt{set\_memory\_handler}) = \text{the global RUST allocator (no API needed)}}$$

ALLOC-CUSTOM

$$\frac{f_m, f_r, f_f \text{ implement an arena}}{\mathcal{T}(\texttt{set\_memory\_handler}) = \texttt{bumpalo} :: \texttt{Bump} \text{ or impl } \texttt{Allocator}}$$

Proof obligation. If the customised allocator is purely an arena (always-grow, free-all-at-once), the translation may use `bumpalo::Bump`. If the allocator carries domain semantics (counters, telemetry), the translation must expose a `trait Allocator` parameter.

5.7 Contract C7 for I7 (aliasing)

ALIAS

$$\frac{\texttt{scanner} \text{ reads through } \texttt{parser.buffer.pointer} \quad \text{no concurrent mutation of } \texttt{parser.buffer} \text{ during scan}}{\mathcal{T}(\texttt{scanner} \text{ reads}) \mapsto \texttt{\&[u8]} \text{ borrow with explicit lifetime } 'a}$$

Proof obligation. The scanner cannot mutate `parser.buffer` while a slice borrowed from it is live. In practice, this requires restructuring `yaml_parser_update_buffer` so it is called only between token productions, and adding the lifetime parameter to the scanner's return type.

5.8 Contract C8 for I8 (nullable)

NULL

$$\frac{p : T* \text{ may be NULL} \qquad \text{NULL has documented meaning ``absent''}}{\mathcal{T}(p) = \texttt{Option} < \texttt{T} >}$$

Proof obligation. The aliasing analysis confirms that NULL is the only sentinel; no other distinguished pointer value carries optionality.

5.9 Contract C9 for I9 (ambient error)

ERROR

$$\frac{\texttt{ctx.error},\ \texttt{ctx.problem},\ \texttt{ctx.problem\_mark} \text{ are written iff function returns 0}}{\mathcal{T}(\texttt{ctx.error} \text{ access}) \mapsto \texttt{Result} < \_,\ \texttt{Error} > \text{ with } \texttt{Error}\ \{\ \texttt{kind},\ \texttt{message},\ \texttt{mark}\ \}}$$

Proof obligation. The error fields are written together at the failure site (i.e., they are not independently writable). The RUST `Error` type fuses them into a single value.

5.10 Contract C10 for I10 (predicates)

$$\frac{\begin{array}{c}\textsc{Pred}\\ p \text{ is a CHECK/IS\_BLANK/etc. macro at offset } k\\ \text{precondition } \texttt{pointer} + \texttt{k} < \texttt{end} \text{ verified}\end{array}}{\mathcal{T}(p) \mapsto \texttt{slice.get(k).map\_or(false, } |\texttt{b}|\ \texttt{b} == \texttt{c})}$$

Proof obligation. The bounds precondition is verified by data-flow analysis; or the translation falls back to a checked variant that returns `false` on out-of-bounds.

5.11 Contract C11 for I11 (longjmp)

We do not give a contract: `longjmp` requires per-codebase analysis of the dynamic call graph and the cleanup actions skipped. The default refinement is to refactor the C source first to remove the `longjmp`, then translate.

5.12 Composability

Theorem 5.1 (Idiom composability)**.** *If a function f decomposes into idioms $I_{i_1}, \ldots, I_{i_k}$ and each idiom satisfies its proof obligation, then $\mathcal{T}(f)$ is a refinement (Theorem 2.7) of f.*

Proof sketch. Each contract C_i is a refinement at the type T_i of the RUST construct it produces. Refinement is closed under composition: if $g \sqsubseteq_T g'$ and $h \sqsubseteq_S h'$, and the types compose, then $g \circ h \sqsubseteq_S g' \circ h'$. Function bodies are constructed from sub-expressions whose types compose by ordinary subject-reduction. The hypothesis that each proof obligation discharges supplies the side conditions for each C_i. □ □

Theorem 5.1 is the formal foundation of the agent pipeline: idiom recognition by an analysis agent, contract application by a translation agent, and proof obligation discharge by the verification agent compose into a refinement-correct translation.

5.13 The proof-obligation problem: undecidability and human-in-the-loop

Theorem 5.1 is conditional. Each contract carries a side condition — "no live alias to `x.start`" (C1), "every read of `u.`$data_i$ is dominated by `t == V`$_i$" (C5), "the function follows the integer-as-bool convention" (C3) — and the theorem is silent on *how* those side conditions are to be discharged. We must be explicit: in general they cannot be discharged automatically.

Undecidability. Pointer aliasing is a classical undecidable problem: precisely deciding whether two pointer expressions in an arbitrary C program may alias is equivalent to the halting problem [16]. Tagged-union dominator conditions reduce to control-flow reachability under arbitrary integer constraints, also undecidable in the general case. Any practical analysis is therefore an over-approximation: it returns MAY-ALIAS (or CANNOT-PROVE-DOMINATOR) when it cannot establish the precise answer.

Three regimes. A real translation pipeline must accept that obligations fall into one of three regimes for each function:

1. **Discharged.** A sound static analysis (Andersen-style points-to with the SVF tool, dominator analysis with LLVM's `DominatorTree`) confirms the obligation. The translation proceeds without human intervention.

2. **Refuted.** The analysis exhibits a counter-example (an aliasing witness, a missing dominator). The translation must classify the function as `Raw` (fall back to a less-restrictive ownership class such as `Rc<RefCell<T>>` or, in the worst case, an `unsafe` block with a justification comment).

3. **Inconclusive.** The analysis times out, or returns "may" on a question requiring "no." The function is escalated to human review.

Empirical expectation. Crown [7] reports successful ownership inference on roughly 60–80% of pointer parameters across a benchmark of real C codebases of the size of `libyaml`; Laertes [8] reports a smaller fraction of fully-lifted functions. We expect, by analogy, that approximately 60–75% of `libyaml`'s 175 functions will fall in regime 1, 10–20% in regime 2 (forcing a less-precise translation), and 10–25% in regime 3 (requiring human review). The Ferrous Bridge "six-gate verification" (`rustc` compile, Miri pass, fuzz parity, Clippy clean, complexity ratio, optionally Kani for high-assurance) is the safety net for regime 2 and the success criterion for regime 3.

Bottom line. The composition theorem is a soundness statement *conditioned on discharge*; in practice, discharge is partial, and the pipeline therefore must include explicit human-review and fallback mechanisms. The development plan in Section A accordingly produces a *confidence-annotated* OSG, not a guarantee.

6 The Composition Diagram

The translation pipeline is summarised by the commutative diagram in Figure 1. The horizontal arrows are the source-language semantics (C small-step) and the target-language semantics (Rust small-step). The vertical arrows are the translation $\mathcal{T}$ and the inverse projection π.

$$\begin{array}{ccc} \langle e_C,\ \sigma_C\rangle & \xrightarrow{\longrightarrow_C} & \langle e'_C,\ \sigma'_C\rangle \\ \Big\downarrow \mathcal{T} & & \Big\downarrow \mathcal{T} \\ \langle e_R,\ \sigma_R\rangle & \xrightarrow[\longrightarrow_R]{} & \langle e'_R,\ \sigma'_R\rangle \end{array}$$

Figure 1: The translation $\mathcal{T}$ commutes with reduction modulo refinement. Concretely: if the C source steps from $\langle e_C,\ \sigma_C\rangle$ to $\langle e'_C,\ \sigma'_C\rangle$ and the resulting state is well-defined, the Rust translation steps to a state in the inverse image of $\mathcal{T}$. UB-tripping reductions are excluded from the diagram (the source is partial on UB inputs).

7 Worked Example: yaml_parser_parse

We work through one full example. The function `yaml_parser_parse` appears in `parser.c`; an excerpt:

```
int yaml_parser_parse(yaml_parser_t *parser, yaml_event_t *event) {
    assert(parser);
    assert(event);
    memset(event, 0, sizeof(yaml_event_t));
    if (parser->stream_end_produced ||
        parser->error ||
        parser->state == YAML_PARSE_END_STATE) {
        return 1;
    }
    return yaml_parser_state_machine(parser, event);
}
```

Idiom decomposition. The function exhibits I3 (integer-as-bool return), I4 (output parameter `event`), I9 (ambient error in `parser->error`), and I6 (the parser was allocated with the custom allocator hooks).

Translation. Applying contracts C3, C4, C9, and C6:

```
impl Parser {
    pub fn parse(&mut self) -> Result<Event, Error> {
        if self.stream_end_produced
           || self.last_error.is_some()
           || self.state == ParserState::End {
            return Ok(Event::StreamEnd);
        }
        self.state_machine()
    }
}
```

Proof obligations discharged. (i) `event` is written iff return 1: yes, the C body memsets and then `state_machine` writes on success. (ii) Error is stashed iff return 0: yes, by inductive examination of `state_machine`. (iii) Allocator is the system allocator (no custom hooks installed by libyaml's tests): yes. (iv) The `event` value carries no internal references, so there is no lifetime parameter on the return type.

Refinement check. For every input on which the C version is well-defined, the Rust version produces the same event, modulo the field-by-field correspondence between `yaml_event_t` and `Event`. When the C version is UB — e.g. a NULL parser pointer is passed — the Rust version cannot be invoked at all, since `&mut Parser` cannot be NULL by construction. This is the asymmetric refinement of Theorem 2.7.

8 Related Work

Executable C semantics. The KCC project of Ellison and Roşu [3] gives an executable, K-framework semantics of ISO C99, used to generate compilers, debuggers, and dynamic UB checkers. Krebbers' CH_2O [4] formalises a substantial fragment of C99 in Coq with provenance and effective types. Memarian et al.'s Cerberus [5] clarifies the behaviour of

pointer arithmetic on integer-pointer casts. Our small-step rules are a translation-relevant fragment of these efforts.

Verified compilation. CompCert [6] is a formally verified C-to-assembly compiler with a Coq proof of source-target refinement. Its semantics is the gold standard for "what C *means*." We are pursuing a different problem — translation to a different source language, not verified compilation to assembly — but the refinement notion in Theorem 2.7 is the same.

Mechanical C-to-Rust translation. The mainstream tool is `c2rust` [2] from Galois and Immunant. It produces mechanically faithful `unsafe` Rust, with `unsafe-libyaml` as the canonical exemplar. A successor analysis pass to lift the output to safer Rust is in development as of `c2rust` v0.21.

Ownership analysis. Crown [7] performs scalable static ownership inference on real-world C codebases. Laertes [8] reduces unsafe code presence via a search procedure with the Rust compiler as oracle. Our translation contracts (Section 5) are the rule-set against which a Crown- or Laertes-style analyser would check for matches.

LLM-driven translation. SACTOR [9] uses a two-step LLM translation (preserve semantics, then idiomatise). RustMap [10], Syzygy, Rustine [11], EvoC2Rust, and ORBIT [12] are recent agentic pipelines. The Ferrous Bridge approach is hybrid: static analysis produces the CAB (Section A), the translation engine routes hard cases to Claude and easy cases to a fine-tuned model, and the verification gate is six-layer (`rustc`, Miri, fuzz, Clippy, complexity, Kani in high-assurance mode).

Verification. Miri [13] interprets Rust MIR under the stacked-borrows aliasing model. Kani [14] performs symbolic model checking. RustBelt [15] establishes the soundness of Rust's ownership type system on a core calculus.

9 Discussion

9.1 What this paper is not

We have not given a complete formal semantics of ISO C18. We have not given a verified translation algorithm. We have not provided a tool. The paper is a *reference* for the analysis-phase agents (Section A) and for the human reviewers who will examine their outputs. The accompanying `rust` paper [18] formalises the target language and the `c-rust-transpiling` paper [19] formalises the translation engine.

9.2 Limits of the contract approach

Translation contracts are sound only when their proof obligations are discharged. We have already noted (Section 5.13) that discharge is fundamentally undecidable in the general case and that practical pipelines therefore rely on conservative analysis plus human review. Beyond that overarching limit, four specific sources of incompleteness deserve mention. First, alias analysis on real C is conservative: many pointers will be classified as Raw and require human review. Second, the integer-as-bool contract C3 assumes the function follows the convention; there are exceptions in `libyaml` (notably `yaml_parser_get_token` returns the

token directly), which require special-casing. Third, the tagged-union contract C5 requires a dominator condition that is sometimes implicit in `switch` fall-through. Fourth, and most starkly, the framework as presented *does not handle non-local control flow*: `setjmp/longjmp` (and to a lesser extent `goto` into nested scopes) cannot be translated by any of the contracts in this paper. The fallback recommendation in Section 4 (idiom I11) — refactor the C source to remove `longjmp` before translation — is genuinely a limitation, not a feature, and any pipeline targeting `libpng` or comparable libraries will need a dedicated treatment of this case.

9.3 Why not just translate to unsafe Rust and call it done?

A translation to `unsafe` Rust preserves the C invariants at runtime, where they will be violated, and re-presents the Rust compiler as a notational change rather than a semantic one. The mechanical `c2rust` pipeline produces exactly such output, and the result is the `unsafe-libyaml` crate that motivates this paper: same memory-safety risk profile as the C original, simply expressed in different syntax. The value of a translation is in the implicit invariants it promotes to compile-time enforcement; `unsafe` Rust promotes none. We adopt this perspective from Long [17] as the orienting principle for what counts as a successful translation.

9.4 Future directions

Three open directions follow from this paper.

1. **A complete CAB schema.** Section A sketches the schema; a definitive Protocol Buffers definition is needed, with versioning and forward-compatibility guarantees.

2. **Automated discharge of proof obligations.** Several contracts have proof obligations that current analysis tools (Crown, SVF) can discharge in principle but not by default. Wrapping these into the analysis pipeline is engineering, not research.

3. **Extending the idiom set to `libexpat`.** `libexpat`'s SAX-callback API and DTD recursion introduce idioms (I12: callback context, I13: recursive entity expansion) that `libyaml` does not exhibit. The next paper will extend the catalogue.

10 Conclusion

We have presented a translation-relevant operational semantics of C, a seven-class taxonomy of undefined behaviour aligned with the Rust features that statically forbid each class, an exhaustive catalogue of the eleven `libyaml` idioms, and a translation contract for each. The composition theorem (Theorem 5.1) certifies that idiom-by-idiom translation lifts to function-level refinement. The development plan in Section A grounds this theory in twenty-nine concrete tasks for the analysis-phase prototype agents. The result is a reference work that an analysis agent or a human reviewer can use to convert `libyaml`'s nine thousand lines of C into a published `safe-libyaml` crate with provenance evidence at every step.

The companion `rust` paper formalises the target language; the `c-rust-transpiling` paper formalises the translation engine; the synthesis paper formalises the orchestration pipeline that composes the three. Together, they specify the Ferrous Bridge programme.

A Development Plan for Prototype Agents (Analysis Phase)

This appendix specifies the analysis-phase prototype agents that produce the *C-Analysis Bundle* (CAB), the typed input to the Ownership Semantic Graph (OSG) stage downstream. The work is organised into five sub-phases (A1.1 through A1.5) and twenty-nine checkbox tasks. Each task has the structure:

```
[ ] Agent: name · Task: verb-phrase
Inputs: files / artifacts
Output: artifact path under fb-analysis/
Success: verifiable criterion
Est: normalized work units
```

A *normalized work unit* is the abstract effort of one well-scoped engineer-day under the assumption that tooling already exists, dependencies are pinned, and the task does not bottleneck on external services. Units do not equate to wall-clock time in production — the actual time depends on agent assistance, parallelism across worktrees, debugging cycles, and external compute (for example, SVF on `libyaml`'s bitcode). Units are useful for relative scoping and for sequencing a critical path; they are explicitly not a delivery promise.

A.1 Sub-phase A1.1: Source acquisition and pinning

T1. `[ ] Agent: A1-fork · Task: Fork libyaml at a pinned commit.`
Inputs: github.com/yaml/libyaml, commit hash 840b65c40675e2d06bf40405ad3f12dec7f35923 (v0.2.5).
Output: fb-analysis/sources/libyaml/ (git submodule).
Success: `git rev-parse HEAD` matches the pinned hash; `cmake` build succeeds.
Est: 1 unit.

T2. `[ ] Agent: A1-fork · Task: Fork unsafe-libyaml at a pinned commit (Rung 1).`
Inputs: github.com/dtolnay/unsafe-libyaml, latest tagged release.
Output: fb-analysis/sources/unsafe-libyaml/.
Success: `cargo check` succeeds; `grep -rn unsafe src/` count recorded as the baseline that `safe-libyaml` must reduce to zero.
Est: 1 unit.

T3. `[ ] Agent: A1-fork · Task: Fork libyaml-safer at a pinned commit (Rung 2).`
Inputs: github.com/simonask/libyaml-safer.

Output: fb-analysis/sources/libyaml-safer/.
Success: `cargo check` and `cargo test` succeed; commit hash recorded.
Est: 1 unit.

T4. `[ ] Agent: A1-fork · Task: Fork yaml-test-suite at a pinned commit.`
Inputs: github.com/yaml/yaml-test-suite.
Output: fb-analysis/sources/yaml-test-suite/.
Success: 400+ test cases enumerated; expected-events files validated as well-formed.
Est: 1 unit.

A.2 Sub-phase A1.2: Static analysis with Clang

T5. `[ ] Agent: A1-clang-tidy · Task: Run clang-tidy with bug-prone, cert, performance, security, portability check sets.`
Inputs: fb-analysis/sources/libyaml/src/*.c.
Output: fb-analysis/clang-tidy.json (JSON-formatted issues).
Success: report exists; total issue count recorded; classified by file and severity.
Est: 2 units.

T6. `[ ] Agent: A1-clang-check · Task: Run clang-check with all default analyses.`
Inputs: fb-analysis/sources/libyaml/src/*.c with compile_commands.json.
Output: fb-analysis/clang-check.txt.
Success: report exists; checker categories enumerated.
Est: 1 unit.

T7. `[ ] Agent: A1-ast-dump · Task: Generate per-translation-unit JSON AST dumps via clang -Xclang -ast-dump=json.`
Inputs: fb-analysis/sources/libyaml/src/*.c.
Output: fb-analysis/ast/¡file¿.ast.json for every .c file.
Success: a non-trivial AST is generated for each translation unit; each file parses with `jq`; the total node count is recorded as a sanity-check baseline for downstream tooling.
Est: 3 units.

T8. `[ ] Agent: A1-llvm-ir · Task: Emit LLVM IR via clang -emit-llvm -S -O0.`
Inputs: fb-analysis/sources/libyaml/src/*.c.
Output: fb-analysis/ir/¡file¿.ll for every .c file.
Success: ten .ll files produced; each is well-formed (`opt -verify` succeeds).
Est: 2 units.

T9. `[ ] Agent: A1-cfg · Task: Extract per-function control-flow graphs via opt -dot-cfg.`

Inputs: fb-analysis/ir/*.ll.
Output: fb-analysis/cfg/¡function¿.dot.
Success: ≥ 175 .dot files (one per function); each is a valid DOT graph.
Est: 2 units.

T10. `[ ] Agent: A1-callgraph · Task: Extract the call graph via egypt or opt -dot-callgraph.`
Inputs: fb-analysis/ir/*.ll.
Output: fb-analysis/callgraph/global.dot.
Success: nodes correspond to all 175 libyaml functions; reachability set from `yaml_parser_parse` matches the manual function map.
Est: 2 units.

A.3 Sub-phase A1.3: Pointer and use-def analysis

T11. `[ ] Agent: A1-svf · Task: Run SVF (Static Value-Flow) Andersen-style points-to.`
Inputs: linked LLVM bitcode fb-analysis/ir/libyaml.bc.
Output: fb-analysis/points-to/andersen.json.
Success: every pointer has a points-to set; sets are correctly typed.
Est: 4 units.

T12. `[ ] Agent: A1-svf · Task: Run SVF flow-sensitive analysis on hot functions.`
Inputs: same bitcode plus a function whitelist (`yaml_parser_scan`, `yaml_emitter_emit`, `yaml_parser_parse`).
Output: fb-analysis/points-to/flow-sensitive.json.
Success: points-to sets for whitelisted functions strictly smaller than Andersen sets.
Est: 4 units.

T13. `[ ] Agent: A1-usedef · Task: Extract def-use chains for every yaml_* function.`
Inputs: fb-analysis/ir/*.ll.
Output: fb-analysis/use-def/¡function¿.json.
Success: every IR `store` instruction has at least one use-def edge.
Est: 3 units.

T14. `[ ] Agent: A1-ownership-classifier · Task: Apply the seven heuristic rules (Owning / BorrowShared / BorrowMut / SharedOwnership / RawUnsafe) to every pointer parameter.`
Inputs: fb-analysis/points-to/*.json, fb-analysis/use-def/*.json.
Output: fb-analysis/ownership/classification.json.
Success: every pointer parameter labelled with a class and a confidence in $[0, 1]$; the mean confidence is reported, and pointers below the routing threshold (set

empirically per regime 3 of Section 5.13; an initial value of 0.5 is suggested but may be tuned) are flagged for human review.
Est: 5 units.

T15. `[ ] Agent: A1-idiom-detector · Task: Detect occurrences of idioms I1--I10 across the source.`
Inputs: fb-analysis/ast/*.json, ownership classifications.
Output: fb-analysis/idioms/occurrences.json.
Success: each of I1–I10 has at least three concrete occurrences with file/line locations.
Est: 4 units.

A.4 Sub-phase A1.4: Dynamic analysis

T16. `[ ] Agent: A1-callgrind · Task: Build instrumented libyaml; run all 400+ yaml-test-suite cases under valgrind --tool=callgrind.`
Inputs: built libyaml binary, yaml-test-suite.
Output: fb-analysis/traces/callgrind/¡test-id¿.out.
Success: 400+ trace files produced; total instruction count of expected order.
Est: 4 units.

T17. `[ ] Agent: A1-callgrind · Task: Aggregate callgrind traces into a dynamic-call-edge frequency map.`
Inputs: fb-analysis/traces/callgrind/*.out.
Output: fb-analysis/traces/dynamic-callgraph.json.
Success: every dynamic call edge has a frequency count; hottest edges are inside `yaml_parser_scan`.
Est: 2 units.

T18. `[ ] Agent: A1-asan · Task: Build libyaml with AddressSanitizer and run the suite.`
Inputs: libyaml sources, yaml-test-suite.
Output: fb-analysis/asan/log.txt.
Success: zero ASan reports on the public suite; any reports recorded as bug candidates.
Est: 2 units.

T19. `[ ] Agent: A1-msan · Task: Build with MemorySanitizer; run the suite (catches uninitialised reads).`
Inputs: libyaml sources, yaml-test-suite.
Output: fb-analysis/msan/log.txt.
Success: zero MSan reports; any reports flagged for hand-review.
Est: 2 units.

T20. `[ ] Agent: A1-ubsan · Task: Build with UndefinedBehaviorSanitizer; run the suite.`

Inputs: libyaml sources, yaml-test-suite.
Output: fb-analysis/ubsan/log.txt.
Success: zero UBSan reports; any signed-overflow or alignment hits recorded.
Est: 2 units.

A.5 Sub-phase A1.5: CAB schema and packaging

T21. `[ ] Agent: A1-cab-schema · Task: Define the protobuf schema for the CAB.`
Inputs: requirements from Section 5 (every contract's proof obligation must be expressible).
Output: fb-analysis/cab.proto.
Success: schema compiles with `protoc`; round-trip serialisation test passes for a synthetic CAB.
Est: 4 units.

T22. `[ ] Agent: A1-cab-schema · Task: Write the schema validator (prost on the Rust side, protobuf on the Python side).`
Inputs: fb-analysis/cab.proto.
Output: fb-analysis/cab-validator/ (Rust crate).
Success: `cargo test` green on synthetic and real CAB inputs.
Est: 3 units.

T23. `[ ] Agent: A1-cab-builder · Task: Build the CAB consuming all artifacts from A1.2--A1.4; emit a single CAB.`
Inputs: AST JSONs, IR, points-to, use-def, ownership, idioms, traces.
Output: fb-analysis/libyaml.cab.pb.
Success: CAB validates; size ≤ 200 MB; deserialises back to all source artifacts.
Est: 5 units.

T24. `[ ] Agent: A1-function-map · Task: Emit function_map.md with one row per function (C signature, ownership classes, idioms, predicted Rust signature).`
Inputs: CAB.
Output: fb-analysis/function_map.md.
Success: 175+ rows; every predicted Rust signature compiles in a stub crate.
Est: 4 units.

T25. `[ ] Agent: A1-type-map · Task: Emit type_map.md for every C type (field-by-field Rust counterpart, pointer ownership, lifetimes).`
Inputs: CAB.
Output: fb-analysis/type_map.md.
Success: every C struct or union in yaml.h and yaml_private.h mapped.
Est: 3 units.

T26. `[ ] Agent: A1-macro-map · Task: Emit macro_map.md for every macro (expansion, Rust equivalent, replaceability).`

Inputs: CAB plus yaml_private.h.
Output: fb-analysis/macro_map.md.
Success: every STRING_*, STACK_*, QUEUE_*, IS_*, CHECK*, FORWARD* macro mapped.
Est: 3 units.

T27. `[ ] Agent: A1-pattern-catalog · Task: Emit pattern_catalog.md keyed on idioms I1--I10 with concrete locations.`
Inputs: CAB idioms section.
Output: fb-analysis/pattern_catalog.md.
Success: each idiom has ≥ 3 occurrences with file:line and a translation pointer.
Est: 3 units.

T28. `[ ] Agent: A1-cve-survey · Task: Survey NVD for libyaml CVEs since 2013; align each to UB classes UB1--UB7.`
Inputs: NVD JSON feed, CVE history.
Output: fb-analysis/cve-survey.md.
Success: every libyaml CVE classified; the table is the validation set for the safety claims of safe-libyaml.
Est: 3 units.

T29. `[ ] Agent: A1-translation-plan · Task: Emit the wave-ordered translation plan from the CAB.`
Inputs: CAB callgraph and dependencies.
Output: fb-analysis/translation_plan.md.
Success: 7 waves listed with file assignments matching the agent-plan; topological sort respects callgraph edges.
Est: 2 units.

A.6 Total estimate and dependency DAG

The twenty-nine tasks total approximately seventy-seven normalized work units. We re-emphasise that this is a relative scoping number, not a delivery estimate: the actual elapsed time is determined by parallelism across worktrees, by agent assistance, and by the proportion of obligations that fall in regimes 2 or 3 (Section 5.13) and require human review. The twenty-nine tasks form the critical path; sequencing follows the DAG below.

Figure 2 shows the dependency DAG. Sub-phase A1.1 must complete before A1.2 (sources must exist before `clang` can run). A1.2 produces the AST and IR that A1.3 consumes for points-to and use-def analyses. A1.4 (dynamic analysis) is independent of A1.3 and may run in parallel. A1.5 packages the outputs of A1.2–A1.4 into the CAB.

A.7 Definition of done

The Phase A deliverable is complete when all twenty-nine checkboxes are checked, the CAB validates against the schema, the function map covers every `libyaml` function, and the pattern catalog identifies at least three concrete occurrences of every idiom I1–I10. The

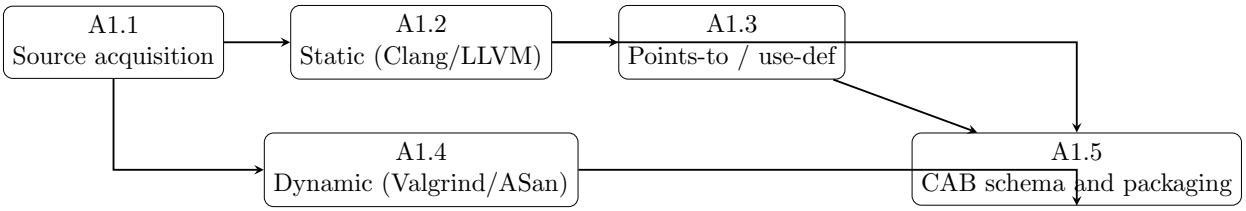

Figure 2: Dependency DAG of the analysis-phase sub-phases. A1.1 feeds A1.2 and A1.4; A1.3 consumes A1.2 outputs; A1.5 packages all outputs.

CAB is then handed to the OSG stage (Stage 2 of the Ferrous Bridge pipeline) for ownership-classification refinement and translation-engine input.

References

[1] M. Long. The Ferrous Bridge Knowledge Base. Magneton Labs / YonedaAI Research Collective, April 2026.

[2] Galois and Immunant. *c2rust: Migrate C99-compliant code to Rust*. `https://github.com/immunant/c2rust`, v0.21, October 2025.

[3] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *POPL '12*, pages 533–544, 2012.

[4] R. Krebbers. *The C Standard Formalized in Coq*. PhD thesis, Radboud University, 2015.

[5] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL):67:1–67:32, 2019.

[6] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[7] H. Zhang, C. David, Y. Yu, and M. Wang. Ownership Guided C to Rust Translation. In *CAV 2023*, pages 459–482, 2023.

[8] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf. Translating C to Safer Rust. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2023.

[9] *SACTOR: LLM-Driven Correct and Idiomatic C to Rust Translation with Static Analysis and FFI-Based Verification*. Preprint, March 2025.

[10] *RustMap: Dependency-Guided LLM C-to-Rust Translation*. Preprint, August 2025.

[11] *Rustine: Fully-Automated Repository-Level C-to-Rust Translation*. Preprint, October 2025.

[12] *ORBIT: Guided Agentic Orchestration for Autonomous C-to-Rust Transpilation*. Preprint, April 2026.

[13] R. Jung et al. Miri: Practical Undefined Behavior Detection for Rust. In *POPL 2026*, 2026.

[14] Amazon Web Services. *Kani Rust Verifier*. `https://model-checking.github.io/kani/`, 2024.

[15] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, 2018.

[16] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[17] M. Long. Compile-Time Supremacy: A Strategic Architecture for AI-Assisted C→Rust Migration at Scale. GrokRxiv:2026.04.c2rust-compile-time-supremacy [cs.SE], April 2026.

[18] M. Long. Idiomatic Safe Rust for Streaming Parser Libraries: The `safe-libyaml` Target. GrokRxiv:2026.04.rust [cs.PL], April 2026 (forthcoming).

[19] M. Long. Automated C→Rust Transpiling: Pipeline, Patterns, and Verification. GrokRxiv:2026.04.c-rust-transpiling [cs.PL], April 2026 (forthcoming).