Hayroll: HARVEST Annotator for Yielding Regions Of Lexical Logic
A Modular Wrapper for Translating C Macros and Conditional Compilation to Rust

Haoran Peng, Baris Kasikci, Gilbert Louis Bernstein, and Michael D. Ernst
University of Washington

2026-06-19, PLDI, Boulder, Colorado, United States

Why Translate C to Rust

- Skipped. Thanks to Cpp2Rust and &inator.
- The C programming language underlies most of the world's software vulnerabilities, mostly due to **memory unsafety**
- About 70% of critical vulnerabilities in Android and Chrome are due to **memory unsafety**
- Rust is designed to be a **memory-safe** language
- A White House report mentions **Rust as a memory-safe** alternative to C/C++
  - Back to the Building Blocks: A Path toward Secure and Measurable Software, Feb 2024, The White House

Existing Transpilers Lose Macros and Configurability

Unpreprocessed C

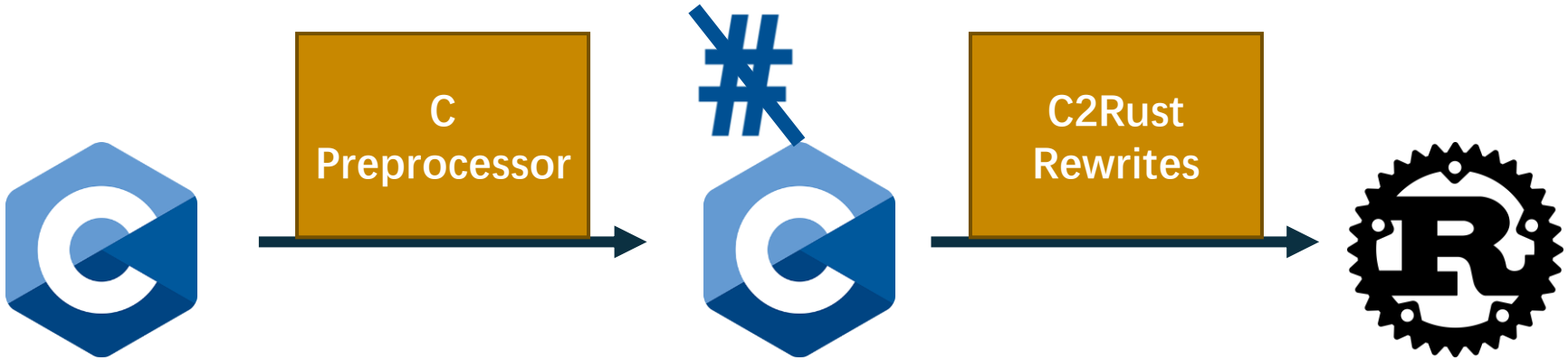
have macro names
configurable compiler flags

Information loss

Rust

lose macro names
fixed compiler flags

Existing Transpilers Run the C Preprocessor in Advance



Macros and Conditional Compilation is an Integral Part of C

- Hayroll translates **C macros** into **Rust functions or macros**
- Hayroll translates **C conditional compilation** into **Rust cfg** attr
- Hayroll is the first and, as of now, the only system that handles this problem

The C Preprocessor Expands Macros and Concretizes Conditional Compilation

```c
float sinhf(float x)
{
#ifdef __LIBMCS_FPU_DAZ
    x *= __volatile_onef;
#endif
    float t, w, h;
    int32_t ix, jx;
    GET_FLOAT_WORD(jx, x);
    ix = jx & 0x7fffffff;
        /* x is INF or NaN */
    if (!FLT_UWORD_IS_FINITE(ix)) {
        return x + x;
    }

    h = 0.5f;
```

```c
float sinhf(float x)
{
    float t, w, h;
    int32_t ix, jx;
    do {
        ieee_float_shape_type gf_u;
        gf_u.value = (x);
        (jx) = gf_u.word;
    } while (0 == 1);
    ix = jx & 0x7fffffff;
    /* x is INF or NaN */
    if (!((ix)<0x7f800000L)) {
        return x + x;
    }

    h = 0.5f;
```

C2Rust Starts from Preprocessed C Code

Cond Stmt Expr

```c
float sinhf(float x)
{
#ifdef __LIBMCS_FPU_DAZ
    x *= __volatile_onef;
#endif
    float t, w, h;
    int32_t ix, jx;
    GET_FLOAT_WORD(jx, x);
    ix = jx & 0x7fffffff;
        /* x is INF or NaN */
    if (!FLT_UWORD_IS_FINITE(ix)) {
        return x + x;
    }

    h = 0.5f;
```

Challenge 1: Source correspondence

Challenge 2: Configuration explosion

-DLIBMCS_FPU_DAZ

```rust
pub unsafe extern "C" fn sinhf(mut x: libc::c_float) -> libc::c_float {
    x *= __volatile_onef;
    let mut t: libc::c_float = 0.;
    let mut w: libc::c_float = 0.;
    let mut h: libc::c_float = 0.;
    let mut ix: int32_t = 0;
    let mut jx: int32_t = 0;
    loop {
        let mut gf_u = ieee_float_shape_type { value: 0. };
        gf_u.value = x;
        jx = gf_u.word as int32_t;
        if !(0 as libc::c_int == 1 as libc::c_int) {
            break;
        }
    }
    ix = jx & 0x7fffffff as libc::c_int;
    if !((ix as libc::c_long) < 0x7f800000 as libc::c_long) {
        return x + x;
    }
```

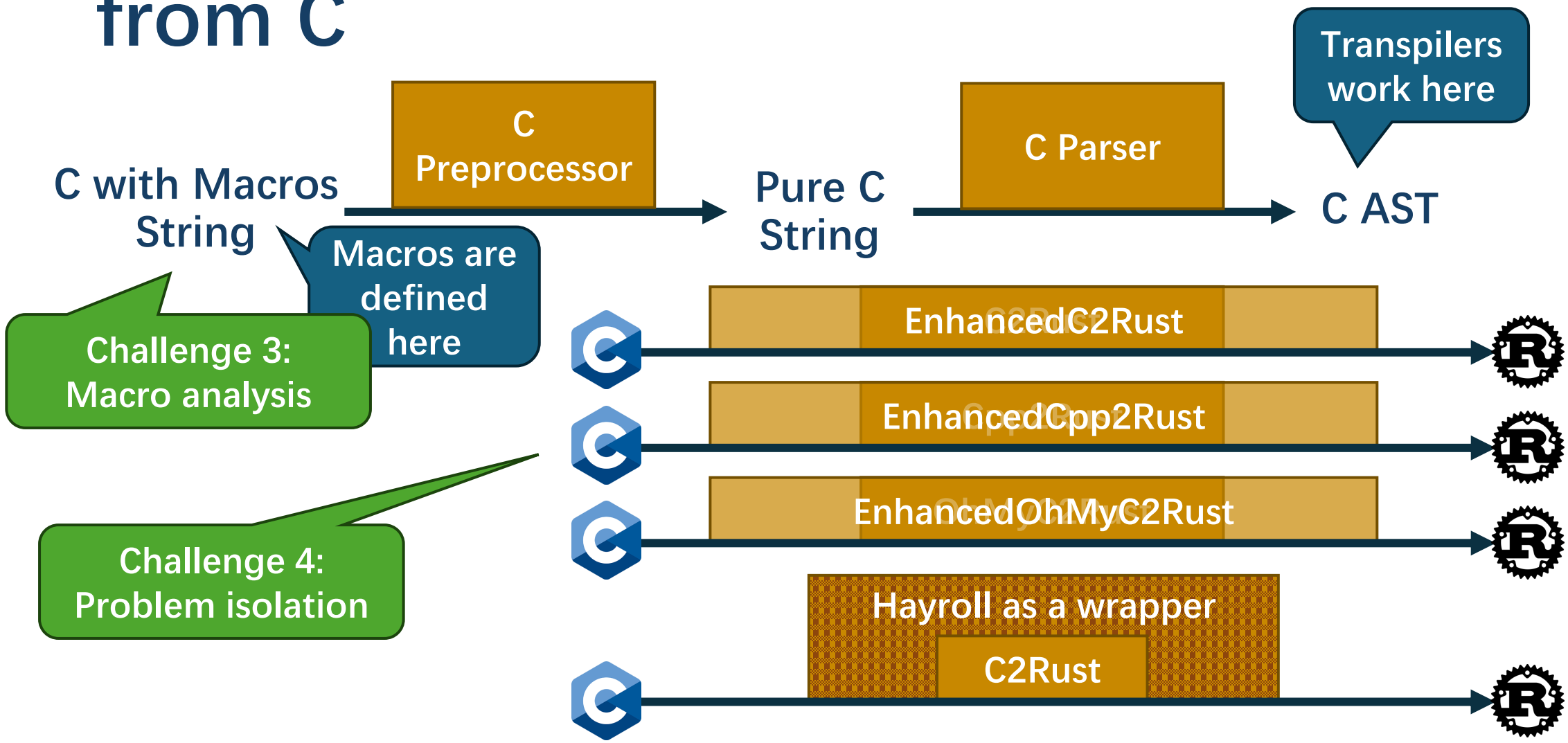
Hayroll Preserves Macros and Conditional Compilation into Rust

Cond Stmt Expr

```c
float sinhf(float x)
{
#ifdef __LIBMCS_FPU_DAZ
    x *= __volatile_onef;
#endif
    float t, w, h;
    int32_t ix, jx;
    GET_FLOAT_WORD(jx, x);
    ix = jx & 0x7fffffff;
        /* x is INF or NaN */
    if (!FLT_UWORD_IS_FINITE(ix)) {
        return x + x;
    }

    h = 0.5f;
```

```rust
pub unsafe extern "C" fn sinhf(mut x: libc::c_float) -> libc::c_float {
    #[cfg(feature = "LIBMCS_FPU_DAZ")]
    (x *= __volatile_onef);
    let mut t: libc::c_float = 0.;
    let mut w: libc::c_float = 0.;
    let mut h: libc::c_float = 0.;
    let mut ix: int32_t = 0;
    let mut jx: int32_t = 0;
    GET_FLOAT_WORD(&mut jx, &mut x);
    ix = jx & 0x7fffffff as libc::c_int;
    if FLT_UWORD_IS_FINITE(&mut ix as *mut int32_t) == 0 {
        return x + x;
    }
    h = 0.5f32;
```

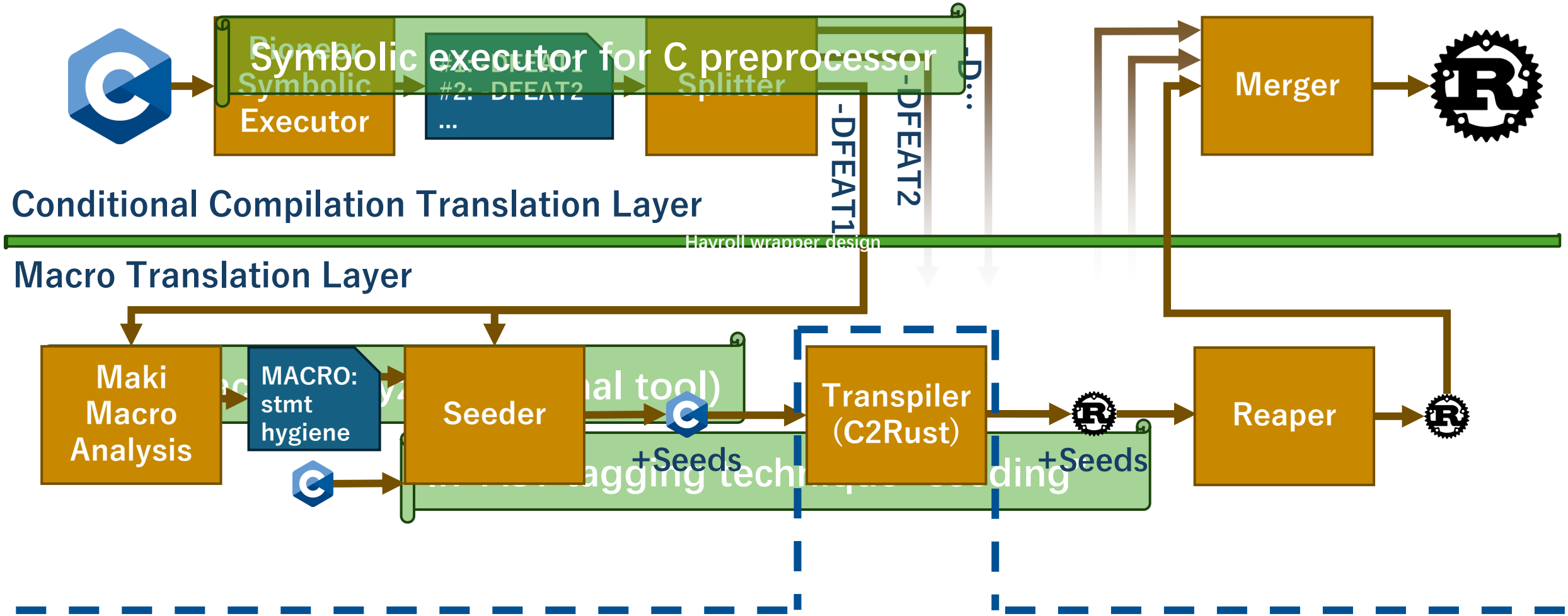
C Preprocessor is a Different Language from C



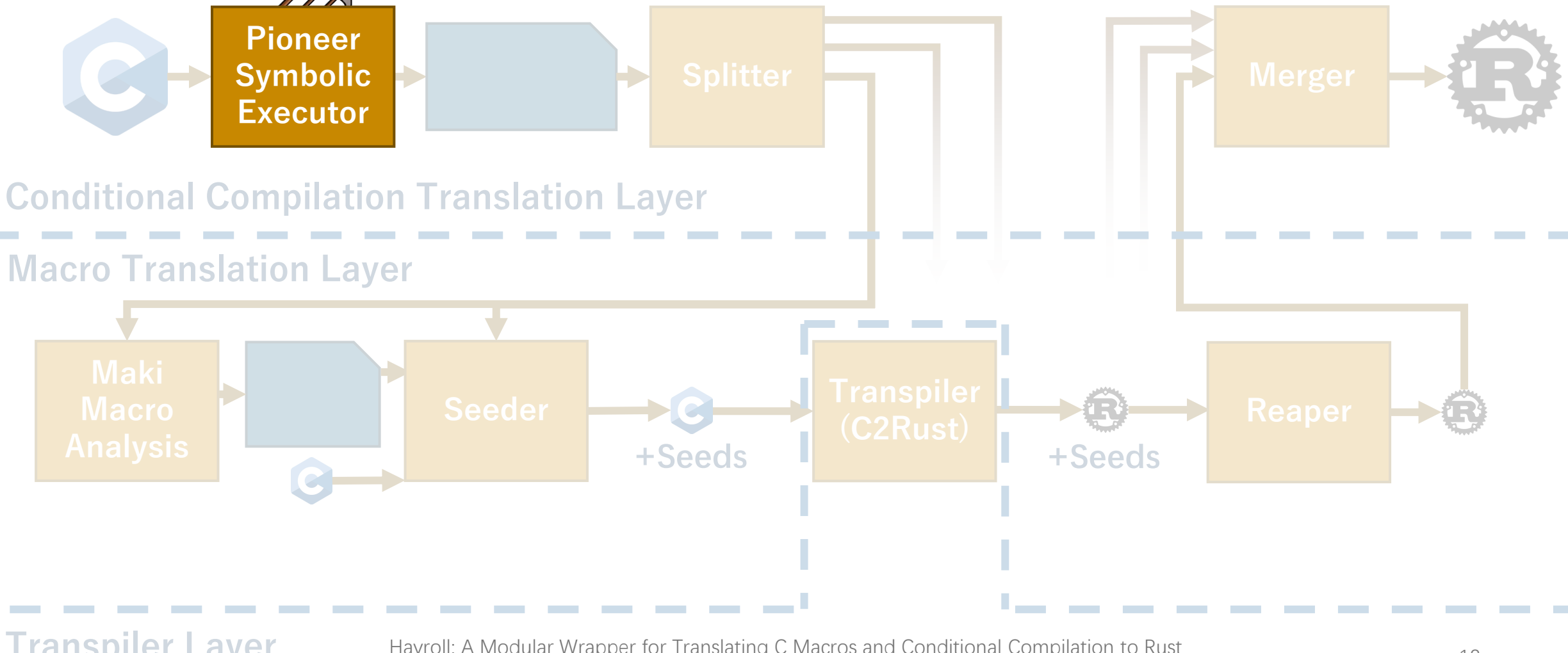
Challenges and Hayroll's Solutions

Challenge	Solution
Problem isolation	Hayroll wrapper design
Macro analysis	Maki macro analyzer (external tool)
Configuration explosion	Symbolic executor for C preprocessor
Source correspondence	In-AST tagging technique "seeding"

Architecture



Architecture



Symbolic Executor: C Conditional Compilation Requires Global Reasoning

To enable this conditional compilation, shall we just define **__LIBMCS_FPU_DAZ**?

```
float sinhf(float x)
{
#ifdef __LIBMCS_FPU_DAZ
    x *= __volatile_onef;
#endif
```

gcc -D__LIBMCS_FPU_DAZ ...
Error: undefined reference to __volatile_onef

Symbolic Executor: C Conditional Compilation Requires Global Reasoning

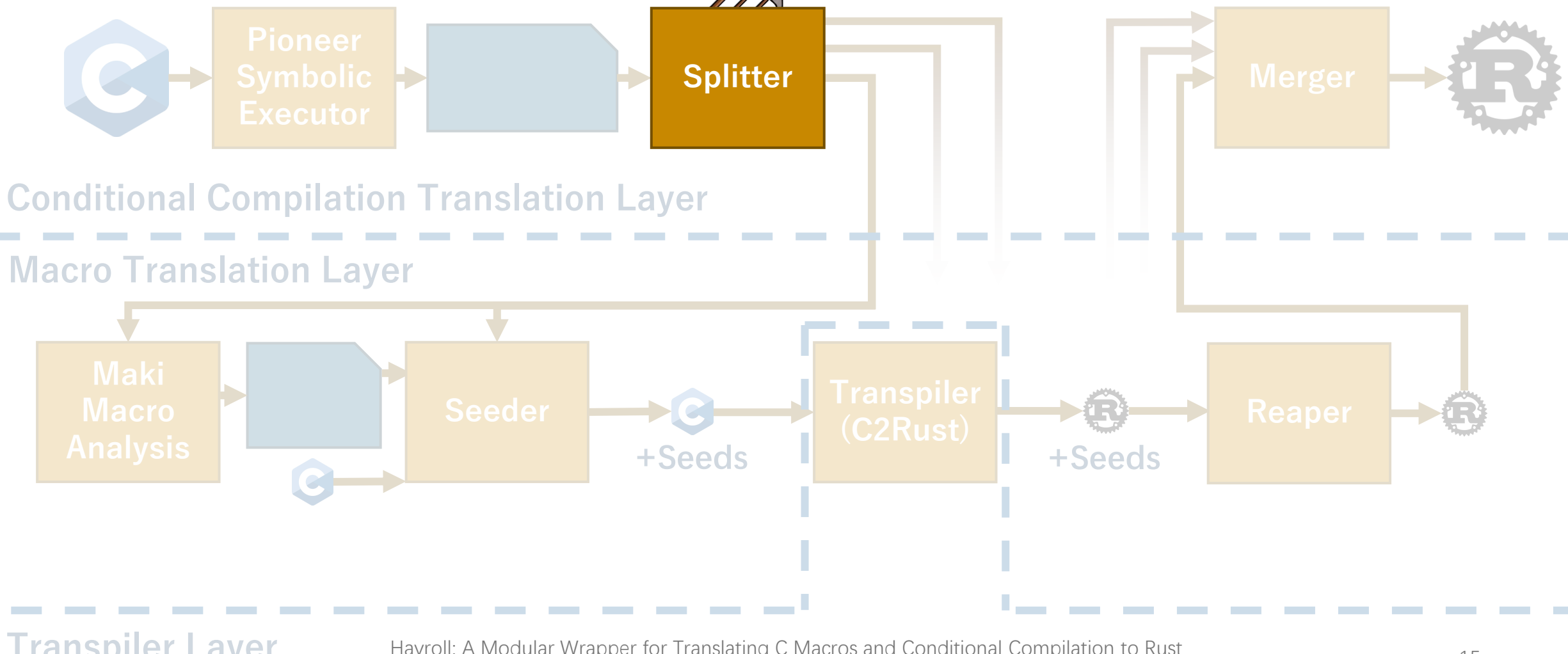
The user should define **LIBMCS_FPU_DAZ** to trigger **underscored __LIBMCS_FPU_DAZ**

```c
// from #include "internal_config.h"
#ifdef LIBMCS_FPU_DAZ
    #define __LIBMCS_FPU_DAZ
    static volatile double __volatile_one = 1.0;
    static volatile float __volatile_onef = 1.0f;
#endif

float sinhf(float x)
{
#ifdef __LIBMCS_FPU_DAZ
    x *= __volatile_onef;
#endif
```

Symbolic Executor output:

- internal_config.h:129:5~135:1 requires **defLIBMCS_FPU_DAZ**
- sinhf.c:13:5~13:26 requires **defLIBMCS_FPU_DAZ**

Architecture



Splitter: Splitting a Configurable Program

into many configuration-specific programs that **together cover every line of code**

```
#ifdef FEAT1
#    ifdef FEAT2
#    endif
#endif
#ifdef FEAT2
#endif
#ifdef FEAT3
#endif
#ifndef FEAT3
#endif
```

All possible combinations

- **(no -D flags)**
- **-DFEAT1**
- **-DFEAT2**
- **-DFEAT3**
- **-DFEAT1 -DFEAT2**
- **-DFEAT1 -DFEAT3**
- **-DFEAT2 -DFEAT3**
- **-DFEAT1 -DFEAT2 -DFEAT3**

Splitter: Splitting a Configurable Program

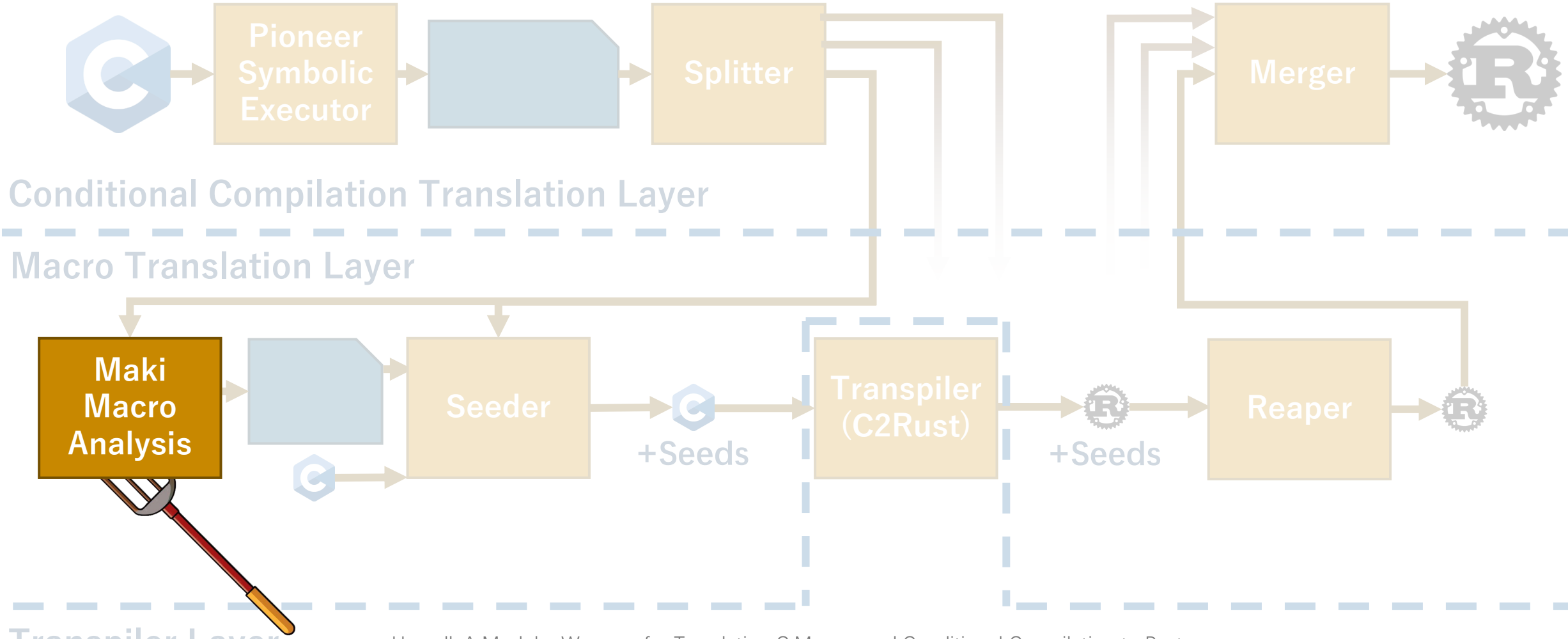
into many configuration-specific programs that **together cover every line of code**

```
#ifdef FEAT1
#    ifdef FEAT2
#    endif
#endif
#ifdef FEAT2
#endif
#ifdef FEAT3
#endif
#ifndef FEAT3
#endif
```

By greedy algorithm (deepest first)

- **`-DFEAT1 -DFEAT2`**
- **`-DFEAT3`**
- **`(no -D flags)`**

Architecture

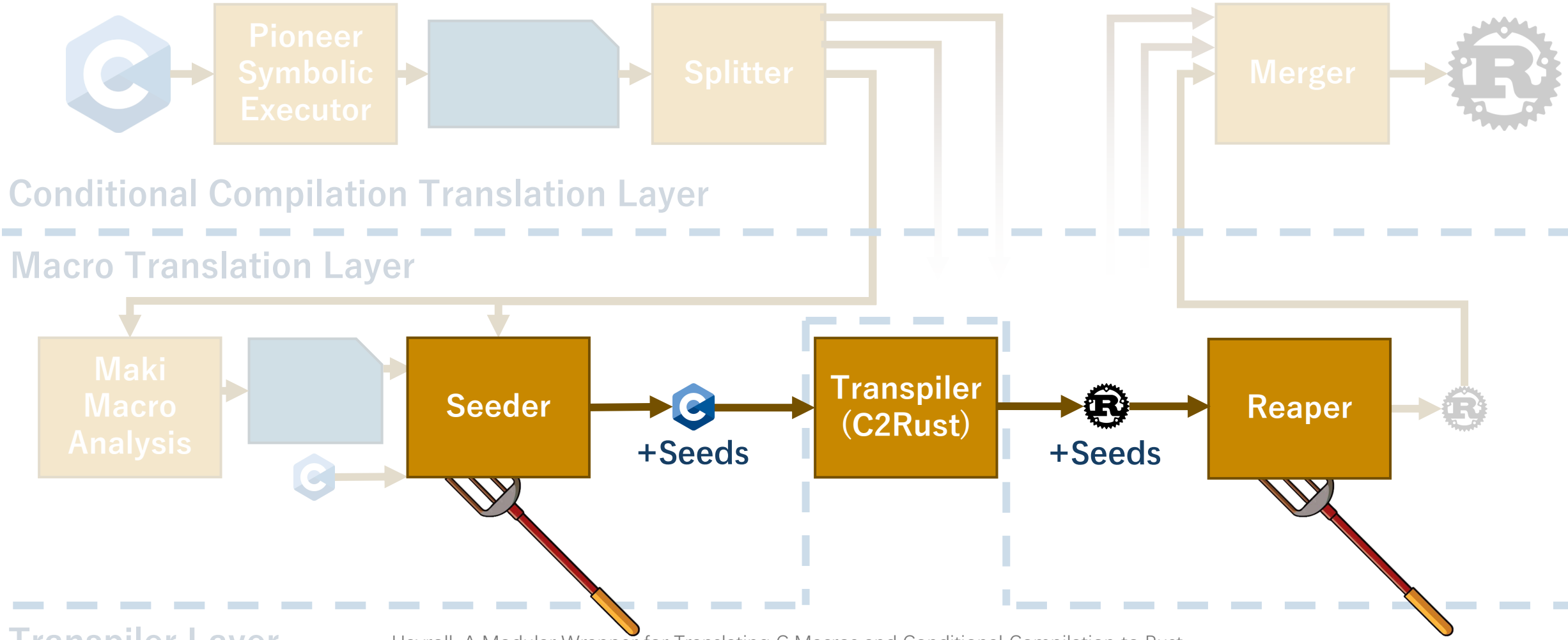


Macro Translatability Properties

- **Leave expanded**
  - Non-syntactical (#define semicolon ;)
  - Non-hygienic (captures local vars)
  - Uses pasting (id##_suffix -> id_suffix)
  - Uses stringification (#name -> “name”)
  - Has function pointer
  - Uses integer constant expression (int a[**N**];)
- **Translate into Rust macro**
  - None of the above properties
  - Has control flow jumps (return, continue, break, goto)
  - Has non-expr arguments (MACRO_FOR(3, {printf(...)}))
  - Argument has side effect (M(i++))
- **Translate into Rust function**
  - None of the above properties

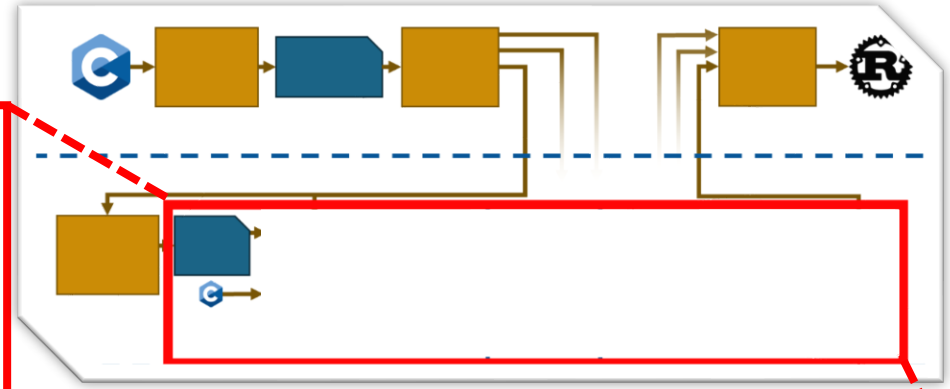
Maki (Hayroll-enhanced version) captures these properties by implementing hooks in Clang’s C preprocessor

Architecture



Seeder+Reaper for Source Correspondence

Architecture



```
GET_FLOAT_WORD(jx, x);
```

Hayroll Seeder

+Seeds

```
"Hayroll Seed Begin";
GET_FLOAT_WORD(jx, x);
"Hayroll Seed End";
```

In-code AST tagging on the C side

C2Rust

+Seeds

```
"Hayroll Seed Begin";
    loop {
        let mut gf_u =
ieee_float_shape_type { value: 0. };
        gf_u.value = x;
        jx = gf_u.word as int32_t;
        if !(0 as libc::c_int == 1 as
libc::c_int) {
            break;
        }
    }
"Hayroll Seed End";
```

Tag retrieval and macro reconstruction on the Rust side

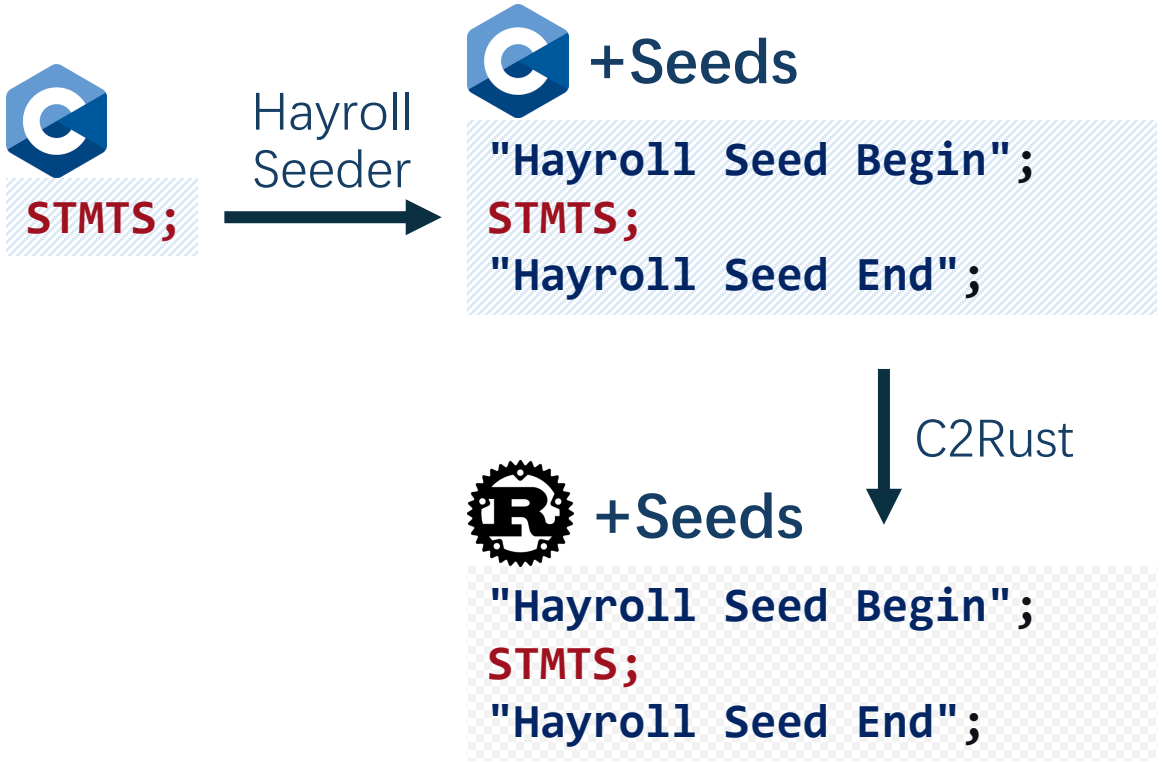
Hayroll Reaper

```
GET_FLOAT_WORD(
    &mut jx, &mut x
);
```

Properties of seeded AST nodes:

- Syntactically preserving
- Semantically preserving
- Carries metadata

Seeding Statement(s)



Seeding Expression

EXPR

Hayroll Seeder

+Seeds

```
(
    "Hayroll Seed" ?
    (EXPR) :
    (*(__typeof__(EXPR)*)(0))
)
```

C2Rust

+Seeds

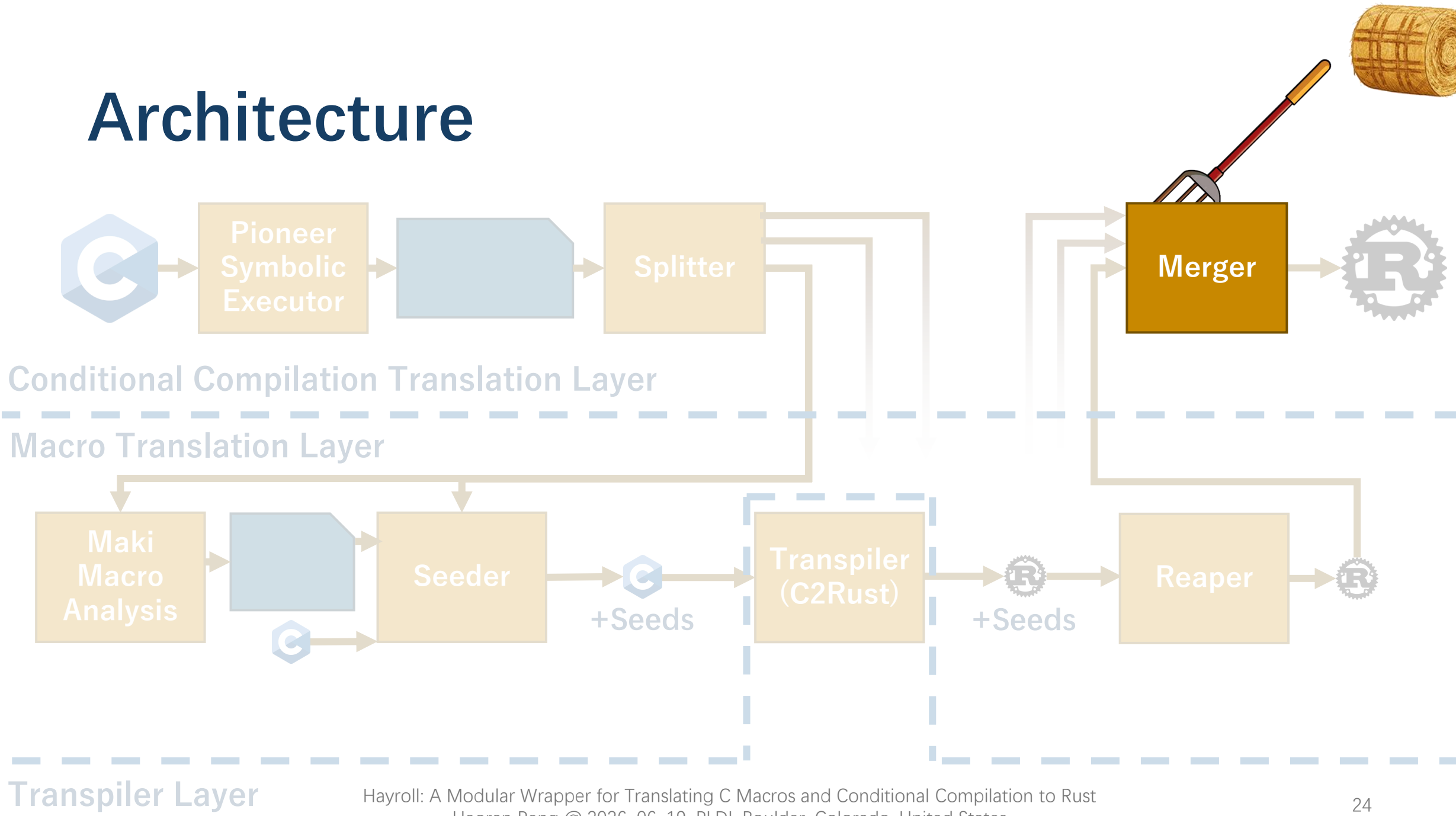
```
if b"Hayroll Seed\0" as *const u8 as libc::c_int != 0 {
    EXPR
} else {
    *(0 as *mut exprtype)
}
```

Translated into an **if expression** in Rust

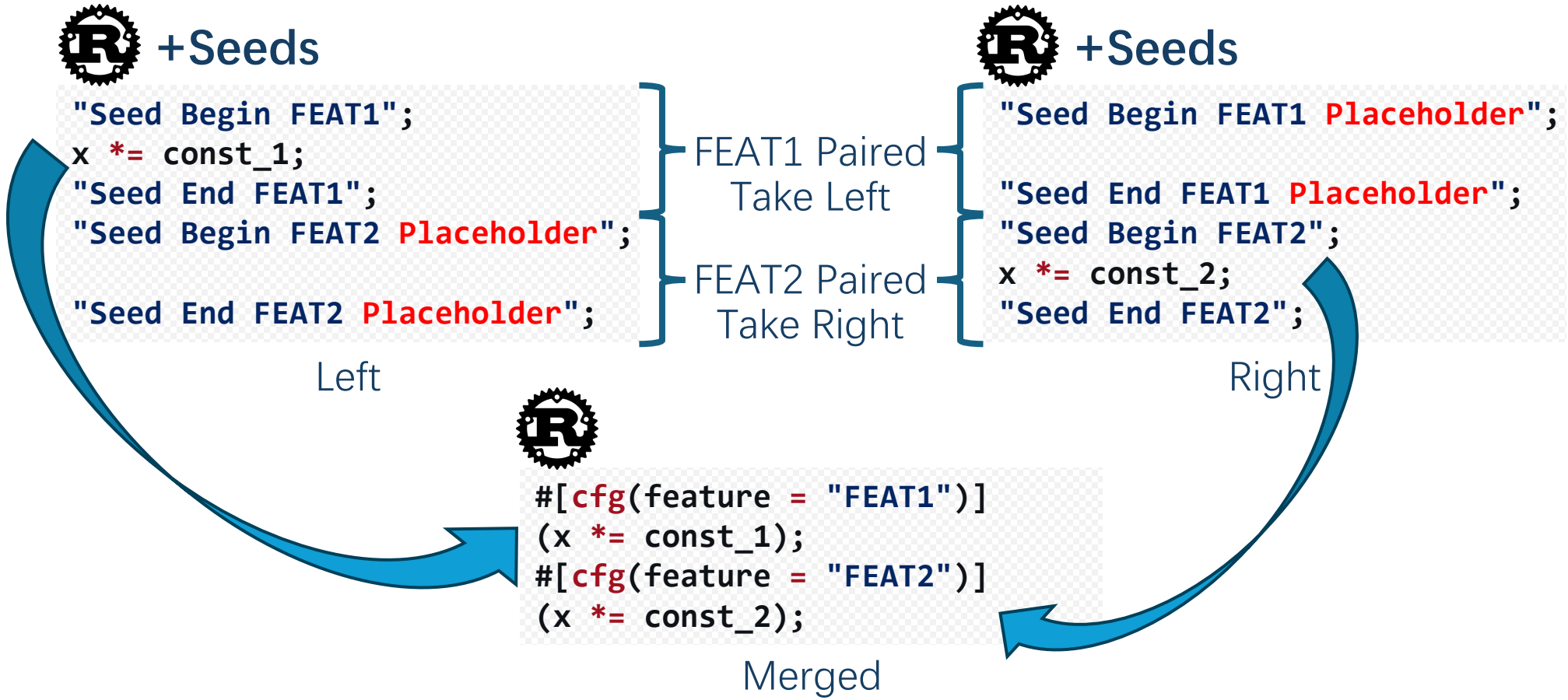
Properties of seeded AST nodes:

- Syntactically preserving
- Semantically preserving
- Carries metadata

Architecture



Merging with Placeholder Seeds



Hayroll Preserves Abstraction and Configurability in Transpilation

Cond Stmt Expr

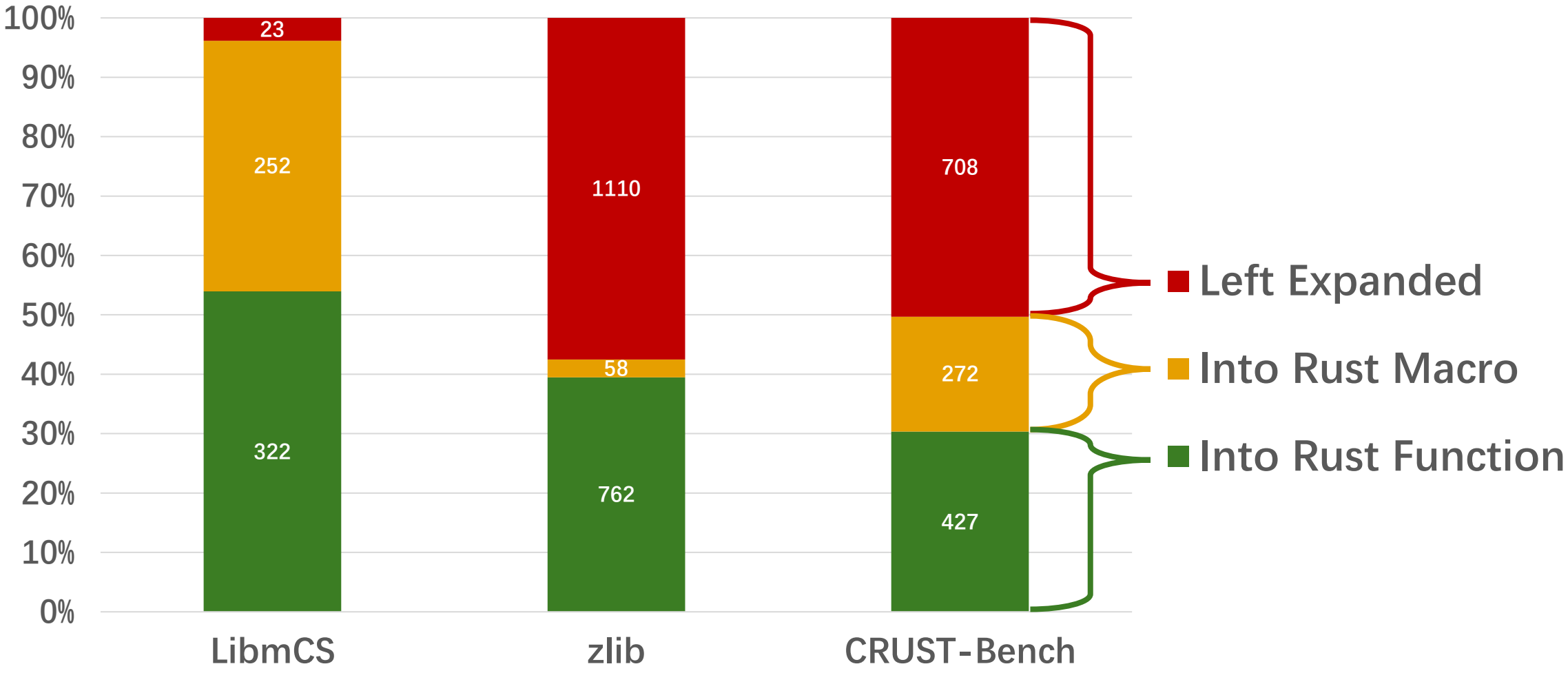
```c
float sinhf(float x)
{
#ifdef __LIBMCS_FPU_DAZ
    x *= __volatile_onef;
#endif
    float t, w, h;
    int32_t ix, jx;
    GET_FLOAT_WORD(jx, x);
    ix = jx & 0x7fffffff;
        /* x is INF or NaN */
    if (!FLT_UWORD_IS_FINITE(ix)) {
        return x + x;
    }

    h = 0.5f;
```

```rust
pub unsafe extern "C" fn sinhf(mut x: libc::c_float) -> libc::c_float {
    #[cfg(feature = "LIBMCS_FPU_DAZ")]
    (x *= __volatile_onef);
    let mut t: libc::c_float = 0.;
    let mut w: libc::c_float = 0.;
    let mut h: libc::c_float = 0.;
    let mut ix: int32_t = 0;
    let mut jx: int32_t = 0;
    GET_FLOAT_WORD(&mut jx, &mut x);
    ix = jx & 0x7fffffff as libc::c_int;
    if FLT_UWORD_IS_FINITE(&mut ix as *mut int32_t) == 0 {
        return x + x;
    }
    h = 0.5f32;
```

Evaluation

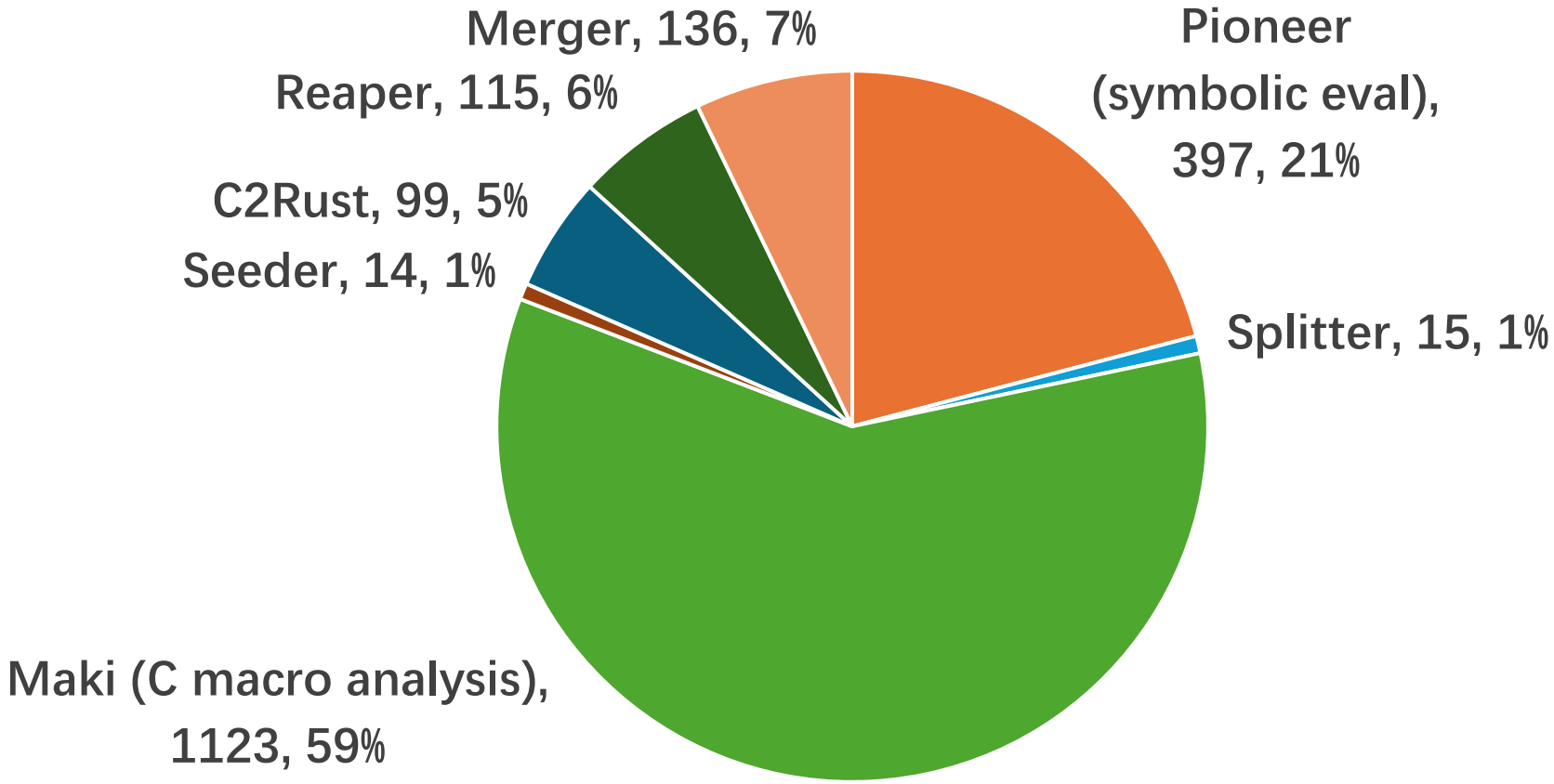
- **Test programs**: LibmCS, zlib, CRUST-Bench
- **Correctness**: Hayroll passes all test cases that C2Rust passes
- **Translator performance**: ~500 LoC/s, ~20x slower than C2Rust
- **Coverage**: See next slide

Fraction of Macros Translated



Per-component Performance (ms/kLoC)

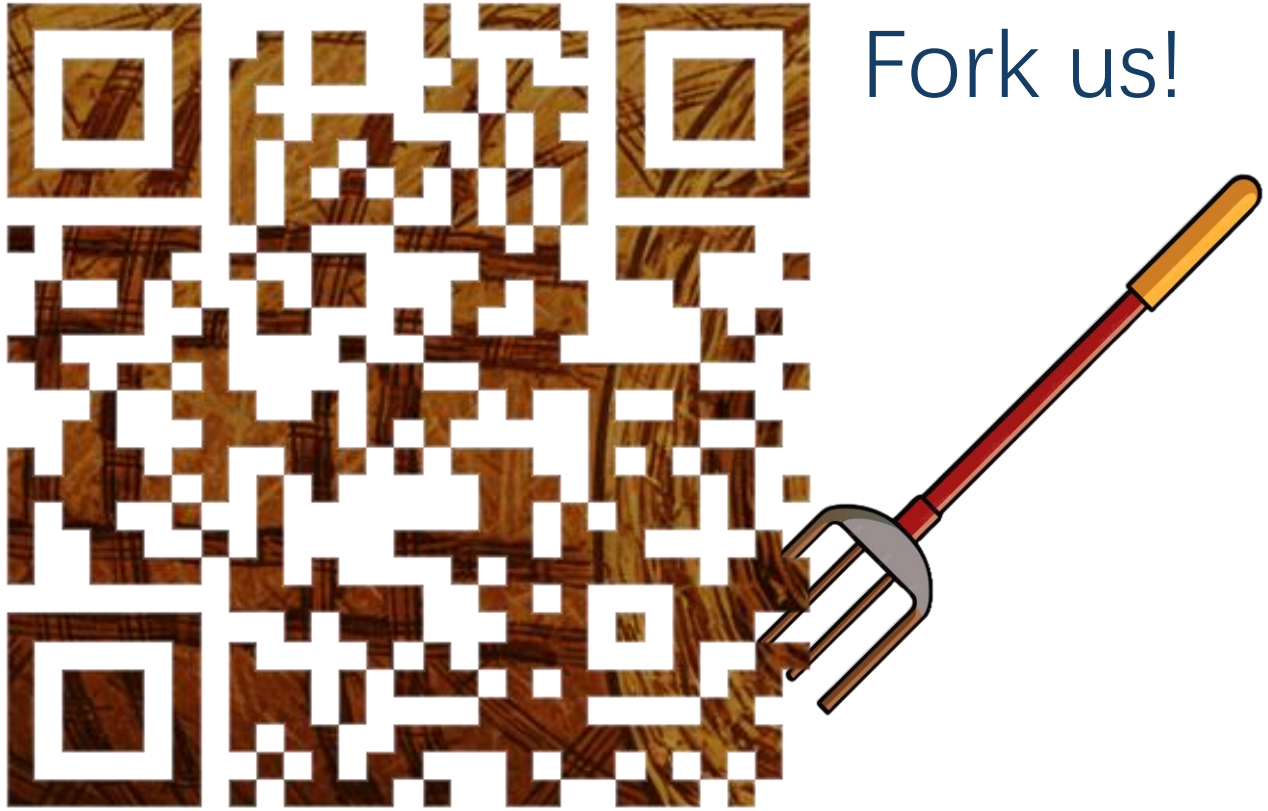
Average over All Programs



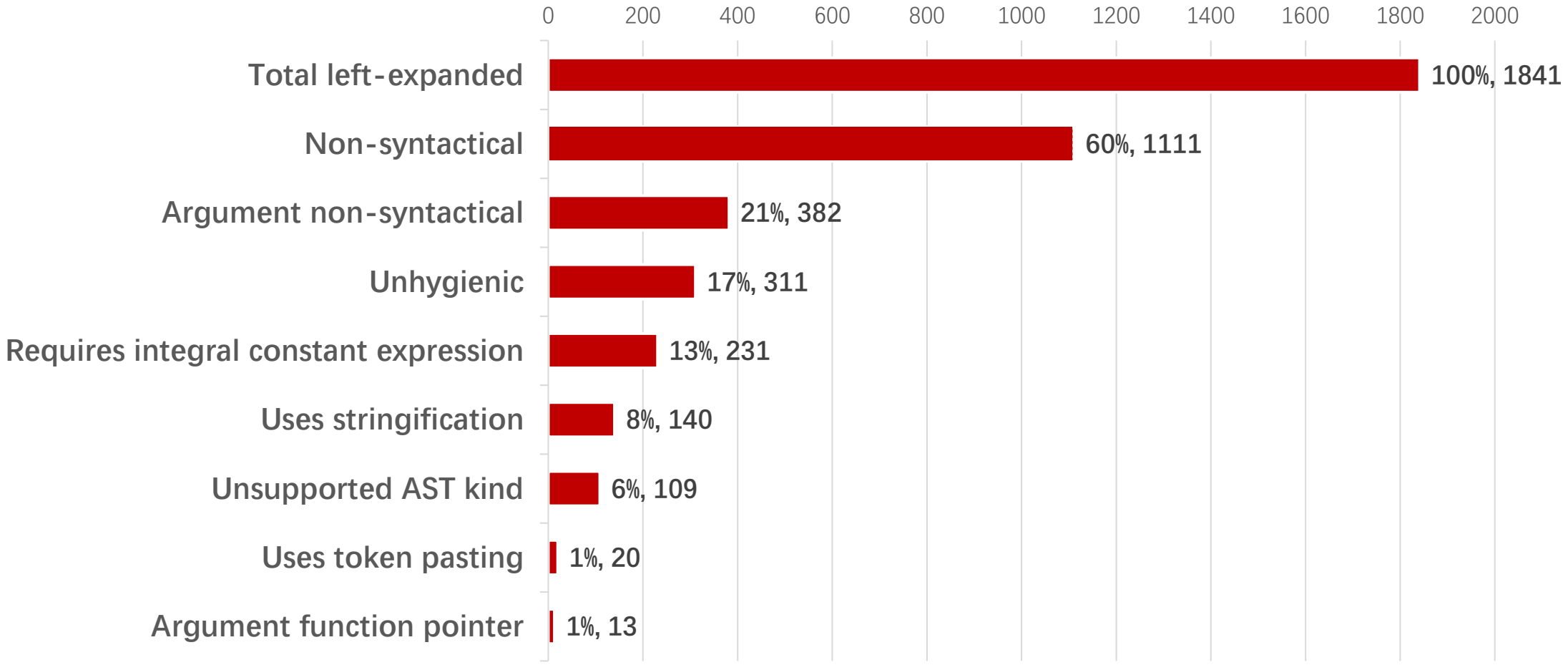
Summary

- Hayroll translates **C macros** into **Rust functions or macros**
- Hayroll translates **C conditional compilation** into **Rust cfg** attr
- Open-sourced at https://github.com/UW-HARVEST/Hayroll
- The C2Rust team is working on integrating Hayroll

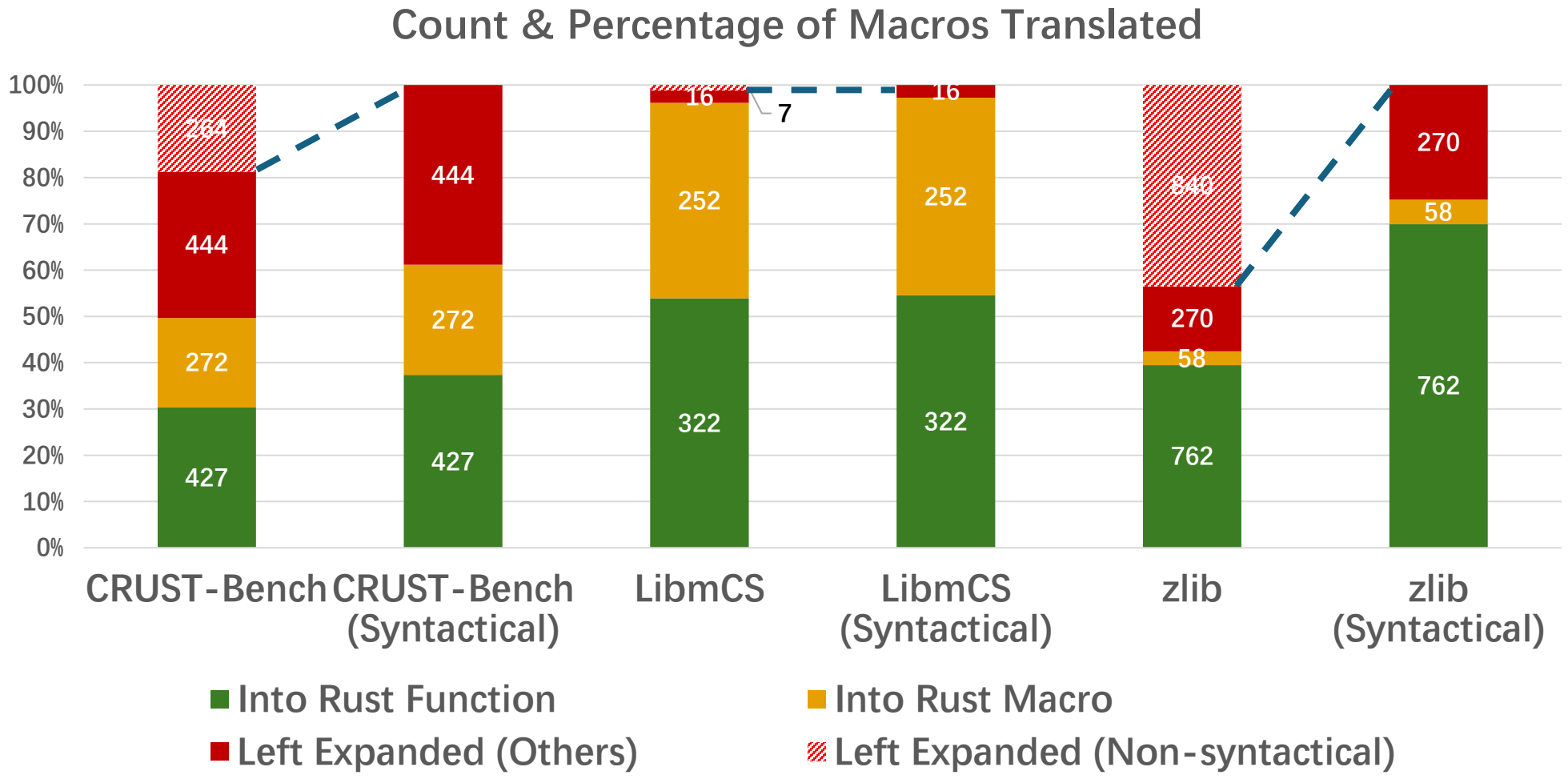
Fork us!



Reasons why Macros are Left Expanded



Effectiveness



Seeding Lvalue Expression

LVALUE

Hayroll Seeder

```
*(
    "Hayroll Seed" ?
    &(LVALUE) :
    (*(__typeof__(LVALUE)*)(0))
)
```

Lemma:

x = y; ⇔ *(&x) = y;

C2Rust

```
*if b"Hayroll Seed\0" as *const u8 as libc::c_int != 0 {
    RVALUE
} else {
    0 as *mut rvaluetype
}
```

Why Translate C to Rust

- The C programming language underlies most of the world's software vulnerabilities, mostly due to **memory unsafety**
- About 70% of critical vulnerabilities in Android and Chrome are due to **memory unsafety**
- Rust is designed to be a **memory-safe** language
- A White House report mentions **Rust as a memory-safe** alternative to C/C++
  - Back to the Building Blocks: A Path toward Secure and Measurable Software, Feb 2024, The White House

Minimap