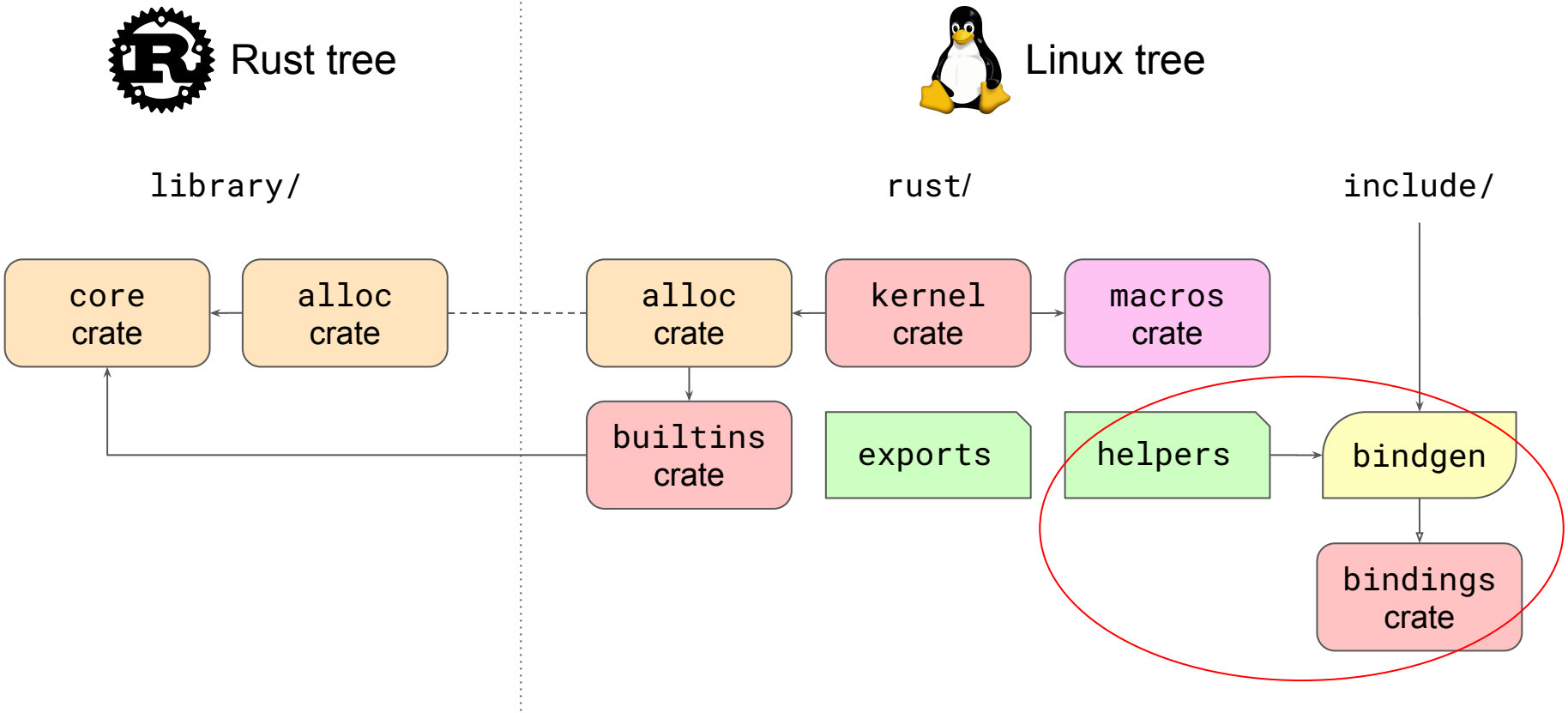
FFI Types and Helpers in Rust-for-Linux

Gary Guo



Credit: Stolen from Miguel

Current Approach

- C types are translated to Rust types using bindgen
- Extern functions are translated to Rust extern function declarations using bindgen
- Inline functions and macros are wrapped with manual C helpers and translated to Rust extern function declarations using bindgen

Issues with our current approach

The current approach works, but:

- C/Rust type mapping are not 1-1
- Excessive unnecessary type casts in Rust code
- Performance loss from calling outlined functions

FFI Types

Integer types: divergence between C and Rust

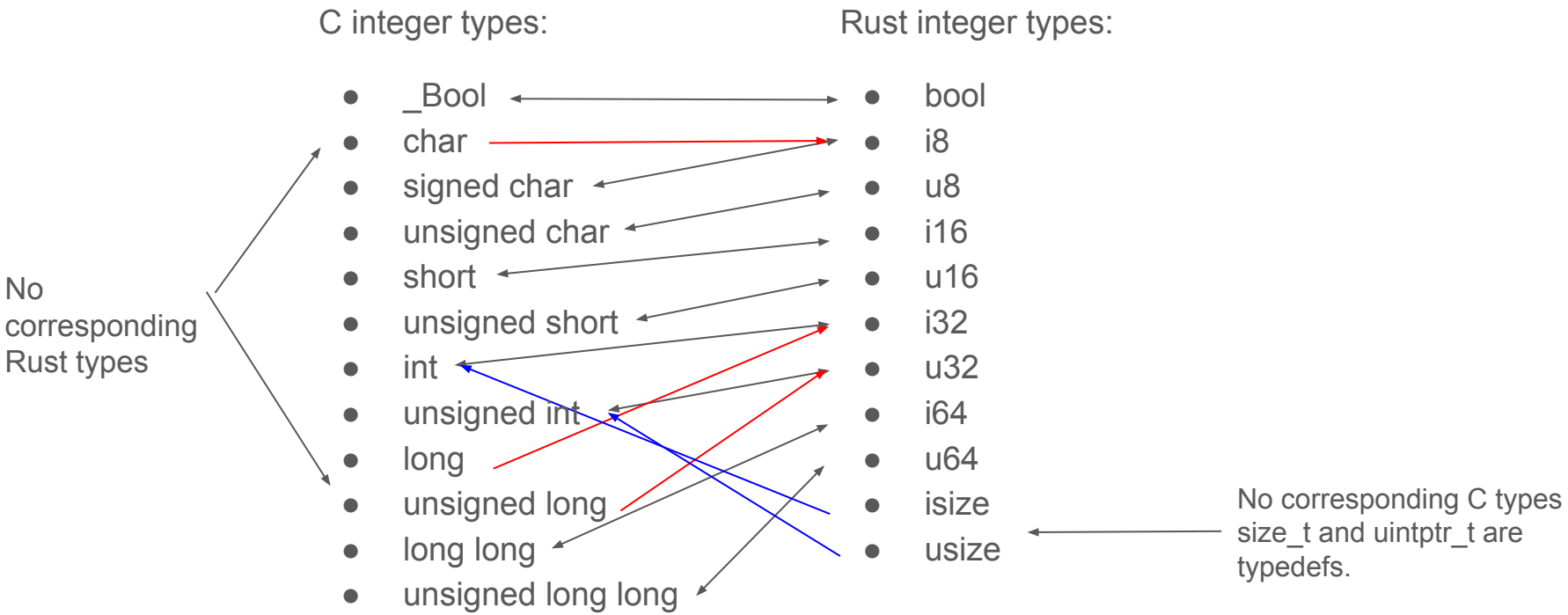
C integer types:

- _Bool
- char
- signed char
- unsigned char
- short
- unsigned short
- int
- unsigned int
- long
- unsigned long
- long long
- unsigned long long

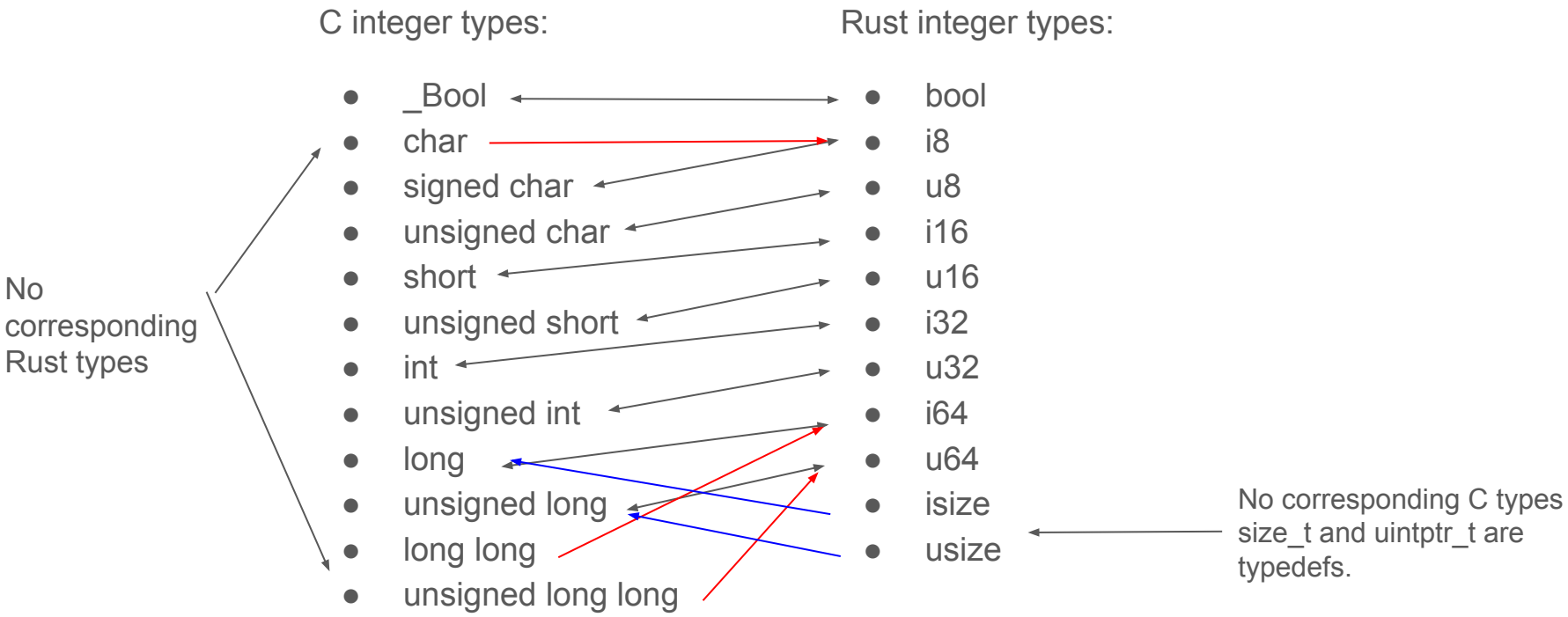
Rust integer types:

- bool
- i8
- u8
- i16
- u16
- i32
- u32
- i64
- u64
- isize
- usize

Integer types: on most 32-bit platforms



Integer types: on most 64-bit platforms



Implication of non-bijection

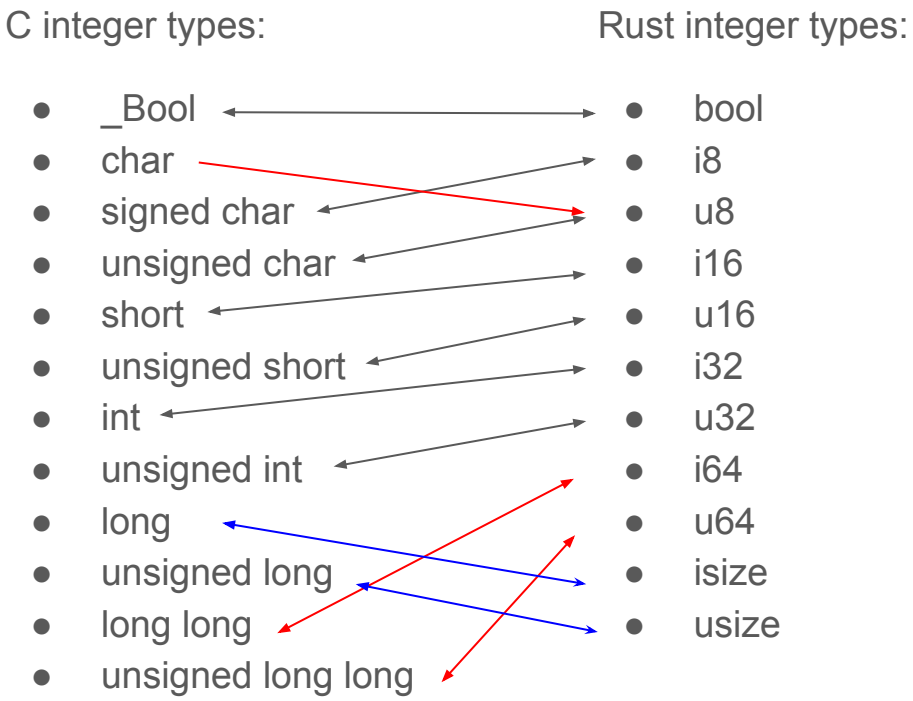
- Translation process is lossy
- CFI/KCFI stops working
  - CFI/KCFI works on actual types, not their sizes
  - Solved by normalizing integer types so that integer types of the same size is treated as the same type.
- Nothing in C translates to isize/usize
  - bindgen workarounds this by treat types named `size_t` specially.
  - Doesn't work for custom `size_t` types, e.g. `__kernel_size_t`

Additional kernel complication

- Kernel defines char to be unsigned unconditionally on all archs
- Rust `core::ffi::c_char` is i8 or u8 depending on arch

- A lot of kernel code uses `long` to mean `intptr_t`.
  - We ended up with a lot of `.try_into()` or `as _`!

A custom bindgen mapping, perhaps?



Some issues

- s64 is defined as long in some cases and long long in other cases:
  - We need to always have it mapped to i64
- size_t seems to be resolved to unsigned int/unsigned long by bindgen.
  - Need to look into what causes this, probably because size_t is treated specially
- Doesn’t work for CHERI, but ignore it for now

FFI Helpers

Usage of helpers

Helpers are used whenever there's a macro or inline function.

```
long rust_helper_PTR_ERR(__force const void *ptr)
{
    return PTR_ERR(ptr);
}
```

A lot of work, and not exactly performant!

Other options?

- Reimplement in Rust
  - Especially unpopular for maintainers
- Transpilation with C2Rust
  - C2Rust is too big to vendor, and is not packaged by any distros to be used as a kernel dependency.
  - Fragile w.r.t. C extensions (e.g. inline asm att syntax)
  - Pinned to rustc nightly-2022-08-08
- Cross-language LTO
  - Slow, resource intensive and sometimes produce broken kernel (LTO support is experimental in kernel)

LTO: Observation

- We only need to inline helpers into Rust call-sites
- Therefore we don’t actually need a global LTO to happen
  - Inlining across two compilation units only is needed
  - Similar to thin local LTO that Rust does for multiple codegen units!

The hack

1. Use clang to generate helpers.ll
2. For each crate
    a. Ask Rust to emit LLVM bytecode
    b. Use llvm-link to combine helpers.ll together with Rust LLVM BC
    c. Feed the combined BC to clang to generate object code
3. Link objects as usual

The hack

1. Use clang to generate helpers.bc
2. For each crate
    a. Ask Rust to emit LLVM bytecode
    b. Use llvm-link to combine helpers.bc together with Rust LLVM BC
    c. Feed the combined BC to clang to generate object code
3. Link objects as usual - Duplicate helper symbols from multiple crates
    a. Use normal linkage causes duplicate symbol
    b. Use weak linkage stops inlining

LLVM linkages

- external (default)
- weak
- linkonce: Weak, but allow discarding if unreferenced
- weak_odr/linkonce_odr:
  - Originally designed for C++ templates
  - Multiple copies of the same symbol can exist, but they must be from a single definition (hence ODR, one-definition rule).
  - In practice, this means: can be inlined, and if not inlined, generated symbols have weak linkage.
  - There doesn’t seem to be a way to generate this from C, though.

The hack - take 2

1. Use clang to generate helpers.ll
    a. Do textual manipulation in helpers.ll to add linkonce_odr everywhere.
    b. Use llvm-as to turn it into helpers.bc
2. For each crate
    a. Ask Rust to emit LLVM bytecode
    b. Use llvm-link to combine helpers.bc together with Rust LLVM BC
    c. Feed the combined BC to clang to generate object code
3. Link objects as usual

The hack - take 2

1. Use clang to generate helpers.ll
    a. Do textual manipulation in helpers.ll to add linkonce_odr everywhere.
    b. Use llvm-as to turn it into helpers.bc
2. For each crate
    a. Ask Rust to emit LLVM bytecode
    b. Use llvm-link to combine helpers.bc together with Rust LLVM BC
    c. Feed the combined BC to clang to generate object code
3. Link objects as usual
4. Inlining didn’t happen

LLVM inlining checks

- llvm/lib/Analysis/InlineCost.cpp has a few checks
  - If target attributes are not compatible, do not inline
    - The target attributes should be compatible, but somehow LLVM is not recognising as such.
    - Probably related to https://github.com/llvm/llvm-project/issues/70002
    - Fix: force inlining with `--ignore-tti-inline-compatible`
  - If no-delete-null-pointer-check setting is not the same, do not inline
    - Fix: force inlining by removing `-fno-delete-null-pointer-checks` when compiling helpers.ll
- Alternative:
  - Use `__always_inline` to bypass all checks.

The hack

- Now everything works!
- Functions are inlined and `rust_helper_` symbols are completely gone.
- Work for both built-in & loadable modules.
- Andreas reports a few percent speedup.
- The con:
  - Similar to LTO, still require matching LLVM version between Clang and Rust

Future possibilities

- Go down the cxx crate route?
  - Rust code specifies the prototype
  - Generate C helper code for inline functions
  - Generate C type compatibility checks for extern functions