Porting Rust to Morello

A safe software layer for a safe hardware layer

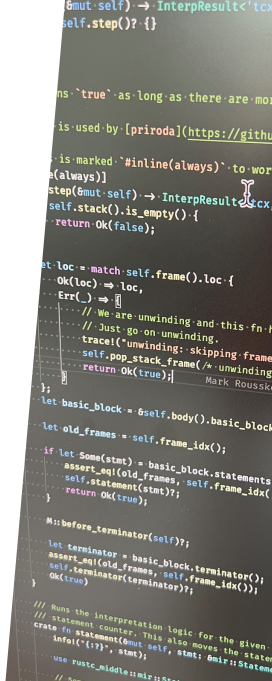
Sarah Harris, **Simon Cooksey**, Mark Batty

{S.E.Harris,S.J.Cooksey,M.J.Batty}@kent.ac.uk

September 2022

Rust

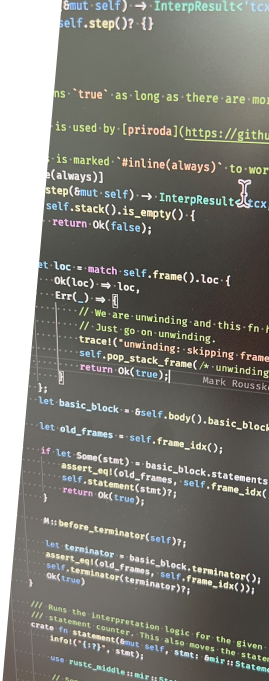
- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.



Rust

- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.

```
fn main() {
  let mut x : [u8; 8] = [0; 8];
  x[9] = 1;
}
```

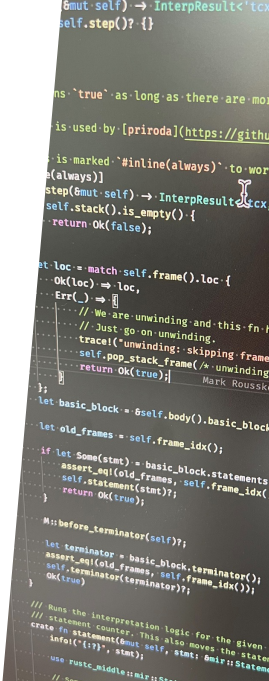


Rust

- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.

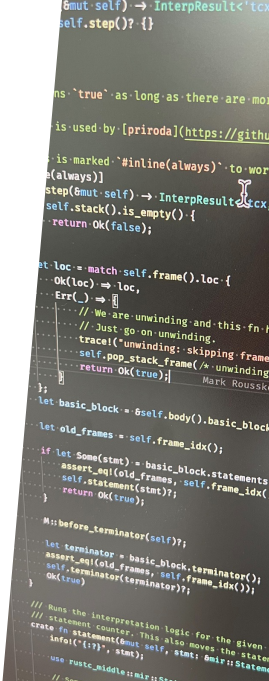
```
fn main() {
  let mut x : [u8; 8] = [0; 8];
  x[9] = 1;
}
```

```
$ rustc ./main.rs -o oob-compile
error: this operation will panic at runtime
--> src/main.rs:3:5
  |
3 |     x[9] = 1;
  |     ^^^^ index out of bounds: the length is 8 but the index is 9
  |
```



Rust

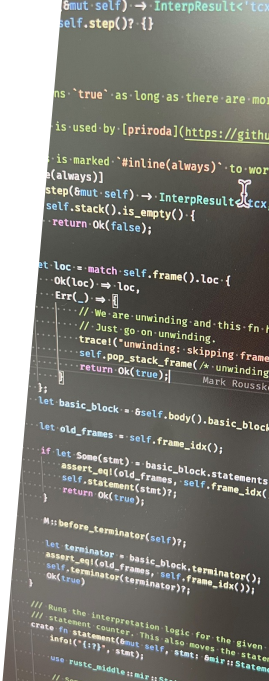
- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.
- Rust is designed to be used in places where C/C++ is used.



Rust

- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.
- Rust is designed to be used in places where C/C++ is used.
- Rust has an escape keyword **unsafe**.

```
fn main() {
    let mut x : [u8; 8] = [0; 8];
    unsafe {
        *x.get_unchecked_mut(9) = 1;
    }
}
```

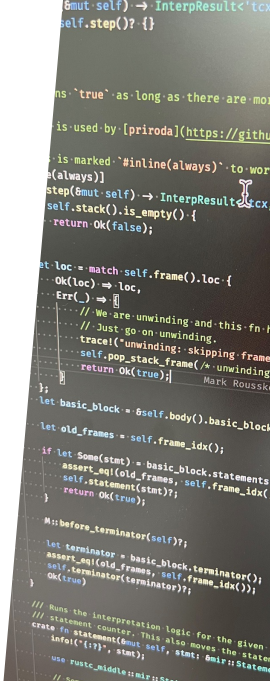


Rust

- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.
- Rust is designed to be used in places where C/C++ is used.
- Rust has an escape keyword **unsafe**.

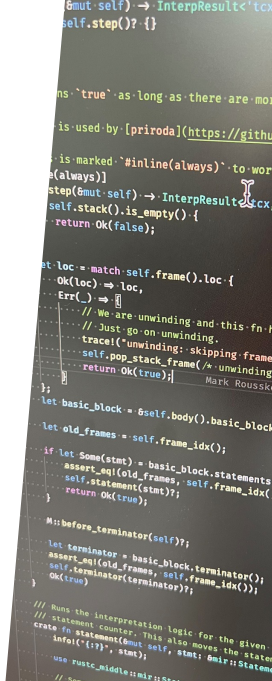
```
fn main() {
    let mut x : [u8; 8] = [0; 8];
    unsafe {
        *x.get_unchecked_mut(9) = 1;
    }
}
```

```
$ ./oob-runtime
Segmentation fault
```



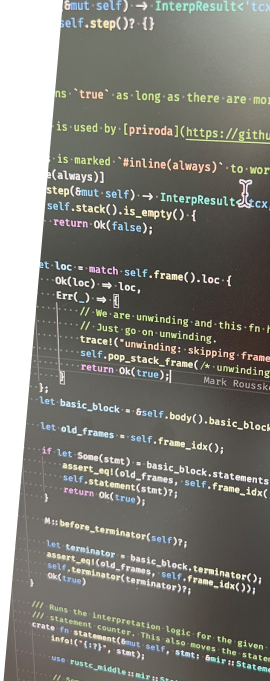
Why port Rust to Morello?

- The guarantees of capabilities complement the guarantees of Rust



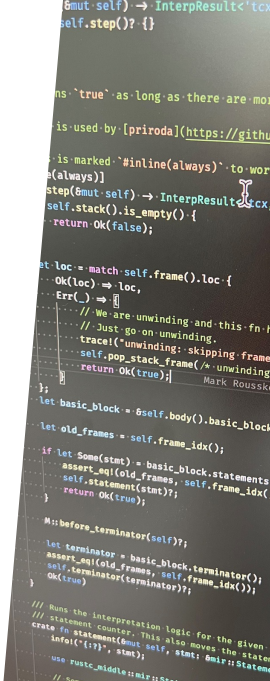
Why port Rust to Morello?

- The guarantees of capabilities complement the guarantees of Rust
- Rust provides compile-time guarantees for `safe` code



Why port Rust to Morello?

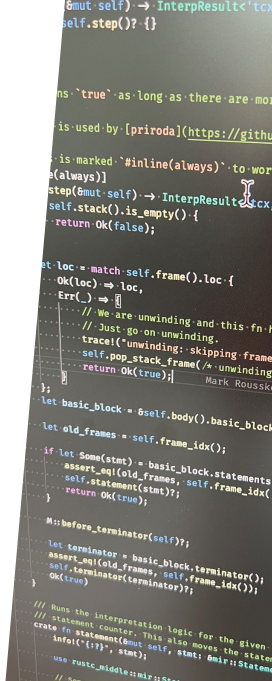
- The guarantees of capabilities complement the guarantees of Rust
- Rust provides compile-time guarantees for `safe` code
- Capabilities provide run-time guarantees for **`unsafe`** code



Why port Rust to Morello?

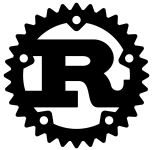
- The guarantees of capabilities complement the guarantees of Rust
- Rust provides compile-time guarantees for `safe` code
- Capabilities provide run-time guarantees for **`unsafe`** code

Digital Security by Design

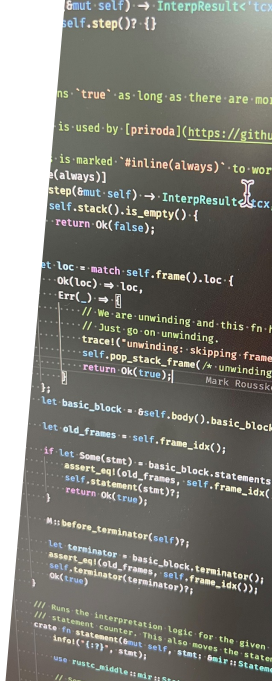


Why port Rust to Morello?

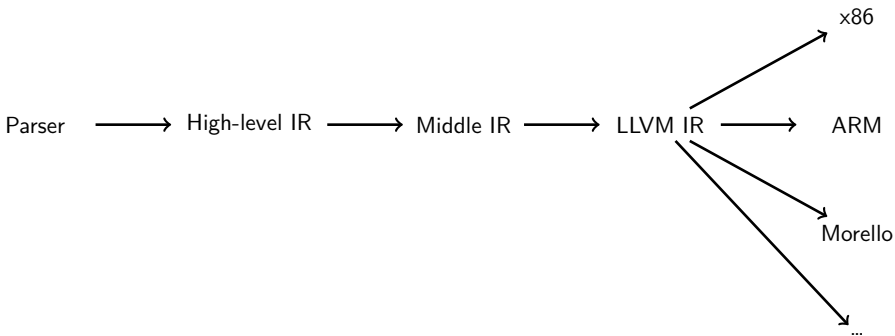
- The guarantees of capabilities complement the guarantees of Rust
- Rust provides compile-time guarantees for `safe` code
- Capabilities provide run-time guarantees for **`unsafe`** code



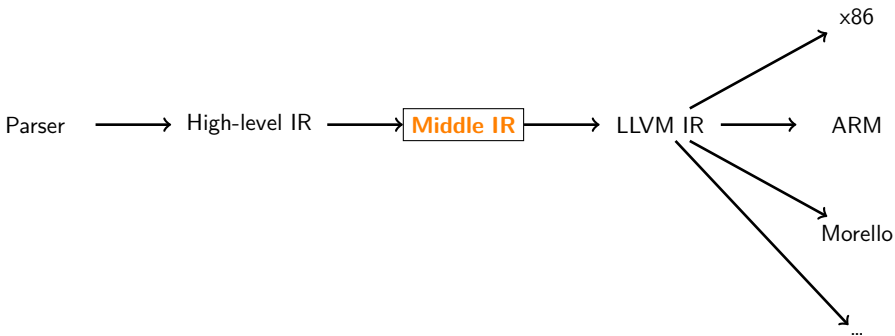
Digital Security by Design



The Rust Compiler



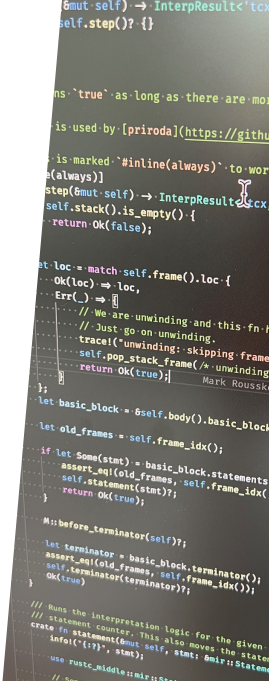
The Rust Compiler



Compiler changes — plumbing

The first task is hooking Rust up with Morello LLVM.

- We added a target, and set the appropriate options
- We hooked up Morello clang as the linker for the Rust compiler
- We extended the Rust target options to allow us to describe object layout differences...

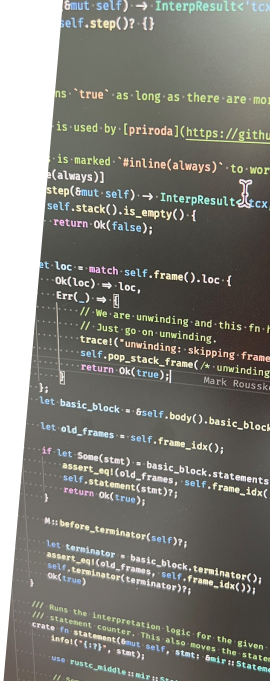


Compiler changes — object layout

Object layout differences, you say?

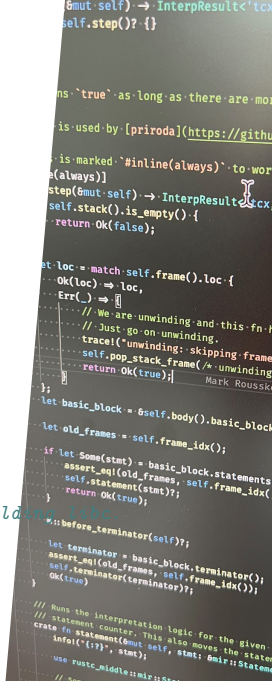
- **`usize`** is a type which must represent the whole range of addresses a pointer can dereference.
- It is used for array indexing, and array bounds.
- We don't want **`usize`** to be 128 bits, memory isn't 128 bit on Morello[†].
- So, we needed to change the layout of a pointer instead.

[†]This approach was explored by Nicholas Sim in his Masters Thesis.



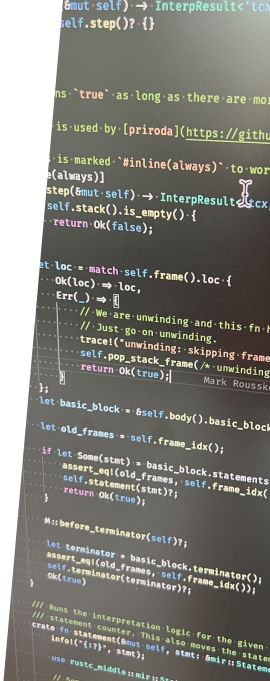
Compiler changes — object layout

```rust
pub fn target() -> Target {
    Target {
        llvm_target: "aarch64-unknown-freebsd".to_string(),
        pointer_range: 64,
        pointer_width: 128,
        data_layout: /* ... */,
        arch: "aarch64".to_string(),
        options: TargetOptions {
            features: "+morello,+c64".to_string(),
            llvm_abiname: "purecap".to_string(),
            max_atomic_width: Some(128),
            // Atomic pointers are supported and converting to integers
            // invalidates capabilities so we *must* use atomic pointers.
            atomic_pointers_via_integers: false,
            // TODO: figure out why this optimisation causes crashes when building libc
            merge_functions: MergeFunctions::Disabled,
            ..super::freebsd_base::opts()
        },
    }
}
```



Compiler changes — constant evaluation

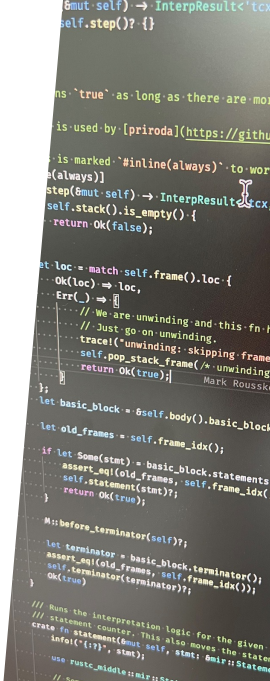
- Rust's IR is interpreted within the compiler to do constant evaluation.
- If it attempts to read uninitialised data that's considered an error.
- We cannot initialise the metadata of these pointers at compile time, so we had to patch up that divide.



Compiler changes — code generation

There are some baked in assumptions in the Rust compiler about valid operations on pointers, for example…

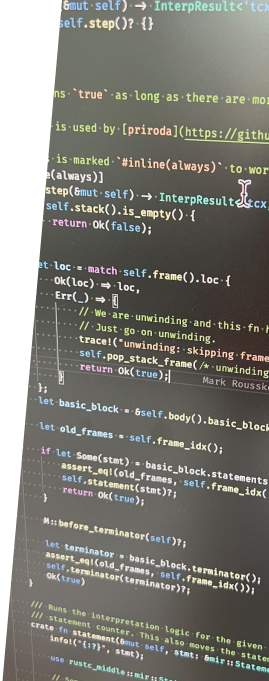
```
if ty.is_unsafe_ptr() {
  // Some platforms do not support atomic operations on pointers,
  // so we cast to integer first.
  let ptr_llty = bx.type_ptr_to(bx.type_isize());
  ptr = bx.pointercast(ptr, ptr_llty);
  val = bx.ptrtoint(val, bx.type_isize());
}
```



Standard library changes

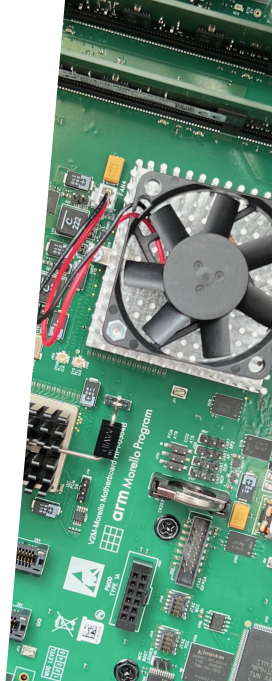
- We're not done here, yet.
- The worst so far has been in a concurrency library which casts pointers to/from integers to tag them with metadata in the lower bits.
- Some bits of the FFI needed some tweaks, integer types being replaced with pointer types.

```
pub unsafe fn cast_from_usize(signal_ptr: usize) -> SignalToken {
    SignalToken { inner: mem::transmute(signal_ptr) }
}
```



What's done

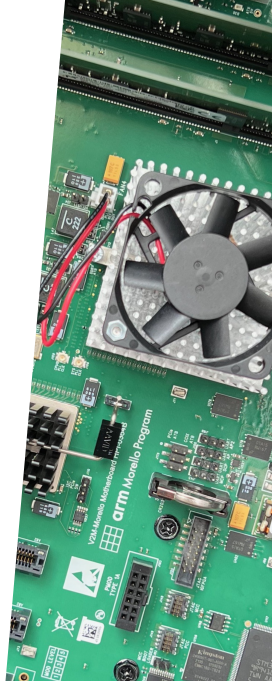
- The compiler emits Purecap code for the Morello machine.
- The `core` part of the standard library is ported, and fairly well tested.
- Various parts of the Rust infrastructure are ported to Morello.
- `std` is ported, but not as thoroughly tested and there are some known bugs to work through.



What's done

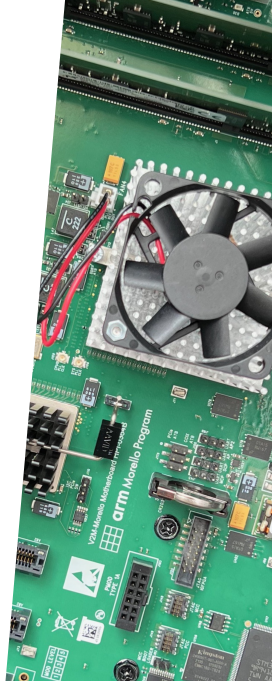
- The compiler emits Purecap code for the Morello machine.
- The `core` part of the standard library is ported, and fairly well tested.
- Various parts of the Rust infrastructure are ported to Morello.
- `std` is ported, but not as thoroughly tested and there are some known bugs to work through.

Demo



Remaining challenges

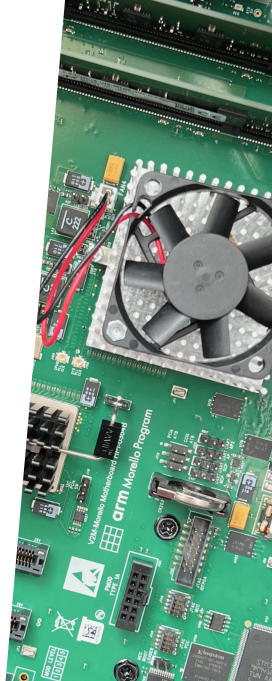
- Porting the rest of the standard library.
- Looking at some code-gen bugs which appear to come from LLVM.
- Port substantial 3rd party software built in Rust to Morello.
- Two optimisations are disabled: SROA, and Function Merging.



What's next?

1. *Performance measurements*: how much does the Rust runtime cost, how much does Morello cost?
   - We have some early work on this. Rust runtime checks look to be on the order of 10% in our testing[†].
2. *Semantic modelling*: how much of the semantics of Morello subsumes the guarantees in safe Rust?
3. *Compiler optimisation*: will the semantics of Morello allow us to remove some dynamic checks from Rust code, but retain Rust's safety properties?
4. *Compiler optimisation*: can we use the Morello prototype's hybrid mode to have zero-overhead statically verified code, and capability checked `unsafe` code?

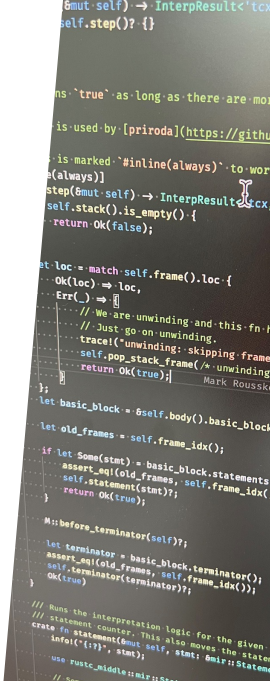
[†]Very early results, not science (yet)!



Formal Verification Goals

- Formalize a semantics for Rust Middle IR (MIR)

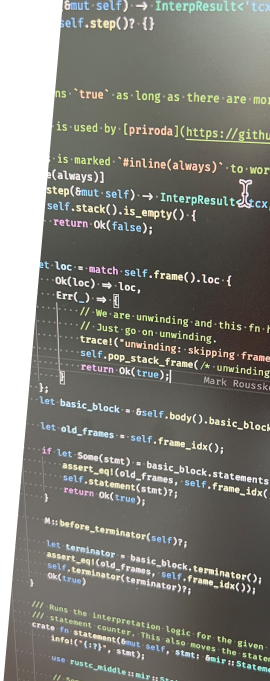
$$[\![P]\!]_{\mathrm{MIR}}$$



Formal Verification Goals

- Formalize a semantics for Rust Middle IR (MIR)
- Prove that compiler optimisations for MIR are sound

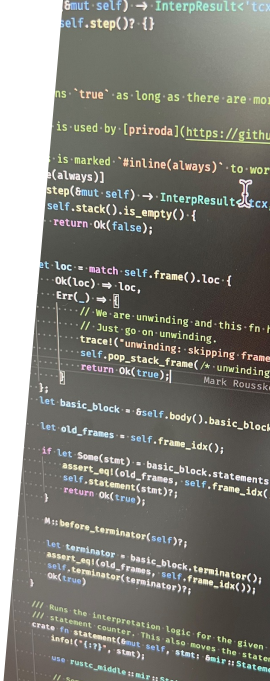
$$[\![\mathrm{opt}(P)]\!]_{\mathrm{MIR}} \subseteq [\![P]\!]_{\mathrm{MIR}}$$



Formal Verification Goals

- Formalize a semantics for Rust Middle IR (MIR)
- Prove that compiler optimisations for MIR are sound
- Prove that MIR is sound with respect to Rust semantics

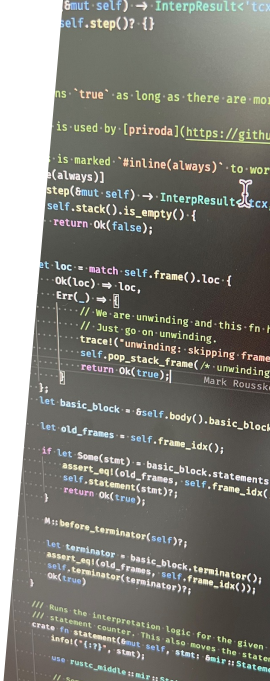
$$[\![\text{rustc}(P)]\!]_{\text{MIR}} \subseteq [\![P]\!]_{\text{Rust}}$$



Formal Verification Goals

- Formalize a semantics for Rust Middle IR (MIR)
- Prove that compiler optimisations for MIR are sound
- Prove that MIR is sound with respect to Rust semantics
- Prove that MIR can be compiled to Morello

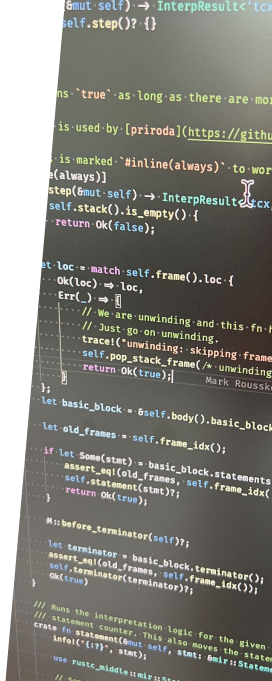
$$[\![\text{rustc}(P)]\!]_{\text{CHERI}} \subseteq [\![P]\!]_{\text{MIR}}$$



Formal Verification Goals

- Formalize a semantics for Rust Middle IR (MIR)
- Prove that compiler optimisations for MIR are sound
- Prove that MIR is sound with respect to Rust semantics
- Prove that MIR can be compiled to Morello
- Prove that any safe Rust code cannot cause a capability fault

$$\text{safe}(P) \implies \nexists X \in [\![\text{rustc}(P)]\!]_{\text{CHERI}}.\text{faulty}(X)$$



Thanks for listening!
Any questions?