

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.

```
    &mut self) → InterpResult<'tcx  
    self.step()? {}  
  
    ns::true` as long as there are mor  
    is used by [priroda](https://github  
    is marked `#inline(always)` to wor  
    e(always)]  
    step(&mut self) → InterpResult<'tcx  
    self.stack().is_empty().{  
    return Ok(false);  
  
    let loc = match self.frame().loc {  
    ... Ok(loc) ⇒ loc,  
    ... Err(_) ⇒ {  
    // We are unwinding and this fn h  
    // Just go on unwinding.  
    trace!("unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding  
    return Ok(true);  
    };  
    let basic_block = &self.body().basic_block  
    let old_frames = self.frame_idx();  
    if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx(  
    self.statements(stmt)?;  
    return Ok(true);  
    }  
    M::before_terminator(self)?;  
    let terminator = basic_block.terminator();  
    assert_eq!(old_frames, self.frame_idx());  
    self.terminator(terminator)?;  
    Ok(true)  
    }  
  
    /// Runs the interpretation logic for the given  
    /// statement counter. This also moves the state  
    crate fn statement(&mut self, stmt: &mir::Statem  
    info!("{:?}", stmt);  
    use rustc_middle::mir::S  
    // See
```

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;  
}
```

```
8mut self) → InterpResult<tcx  
self.step()? {}  
  
ns `true` as long as there are mor  
is used by [priroda](https://github  
is marked `#inline(always)` to wor  
e(always)]  
step(&mut self) → InterpResult<tcx  
self.stack().is_empty() {  
    return Ok(false);  
}  
  
et loc = match self.frame().loc {  
    Ok(loc) ⇒ loc,  
    Err(_) ⇒ {  
        // We are unwinding and this fn h  
        // Just go on unwinding.  
        trace!("unwinding: skipping frame  
        self.pop_stack_frame(/* unwinding  
        return Ok(true);  
    }  
};  
let basic_block = &self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx(  
    self.statements(stmt)?;  
    return Ok(true);  
}  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(&mut self, stmt: &mir::Statem  
    info!("{:?}", stmt);  
    use rustc_middle::mir::S  
    // See
```

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  
|
```

```
8mut self) -> InterpResult<tcx  
self.step()? {}  
  
ns `true` as long as there are mor  
is used by [priroda](https://github  
is marked `#inline(always)` to wor  
e(always)]  
step(8mut self) -> InterpResult<tcx  
self.stack().is_empty() {  
return Ok(false);  
  
et loc = match self.frame().loc {  
Ok(loc) => loc,  
Err(_) => {  
    // We are unwinding and this fn h  
    // Just go on unwinding.  
    trace!("unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding  
    return Ok(true);  
};  
let basic_block = 8self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx(  
    self.statements(stmt)?;  
    return Ok(true);  
}  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(8mut self, stmt: 8mir::Statem  
    info!("{:?}", stmt);  
use rustc_middle::mir::Statem  
// See
```

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.

```
    &mut self) → InterpreterResult<'tcx, ...  
    self.step()? {}  
  
    ns: 'true' as long as there are more  
    is used by [priroda](https://github.com/priroda/priroda)  
    is marked '#inline(always)' to work  
    e(always)]  
    step(&mut self) → InterpreterResult<'tcx,  
    self.stack().is_empty() {  
    return Ok(false);  
}  
  
let loc = match self.frame().loc {  
    Ok(loc) ⇒ loc,  
    Err(_) ⇒ {  
        // We are unwinding and this fn has  
        // Just go on unwinding.  
        trace!("unwinding: skipping frame");  
        self.pop_stack_frame(/* unwinding */);  
        return Ok(true);  
    };  
    Mark Rousskin  
let basic_block = &self.body().basic_block;  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx());  
    self.statements(stmt)?;  
    return Ok(true);  
}  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(&mut self, stmt: &mir::Statement,  
    info!("[{:?}]", stmt);  
    use rustc_middle::mir::TerminatorKind;  
    // See
```

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;  
    }  
}
```

```
8mut self) → InterpResult<'tcx  
self.step()? {}  
  
ns `true` as long as there are mor  
is used by [priroda](https://github  
is marked `#inline(always)` to wor  
e(always)]  
step(&mut self) → InterpResult<'tcx  
self.stack().is_empty() {  
    return Ok(false);  
}  
  
let loc = match self.frame().loc {  
    Ok(loc) ⇒ loc,  
    Err(_) ⇒ {  
        // We are unwinding and this fn h  
        // Just go on unwinding.  
        trace!("unwinding: skipping frame  
            self.pop_stack_frame(/* unwinding  
            return Ok(true);  
    }  
};  
let basic_block = &self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx(  
        self.statements(stmt)?;  
    return Ok(true);  
}  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(&mut self, stmt: &mir::Statem  
    info!("{:?}", stmt);  
    use rustc_middle::mir::S  
    // See
```

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
```

```
8mut self) → InterpResult<'tcx  
self.step()? {}  
  
ns `true` as long as there are mor  
is used by [priroda](https://github  
is marked `#inline(always)` to wor  
e(always)]  
step(8mut self) → InterpResult<'tcx  
self.stack().is_empty() {  
    return Ok(false);  
}  
  
let loc = match self.frame().loc {  
    Ok(loc) ⇒ loc,  
    Err(_) ⇒ {  
        // We are unwinding and this fn h  
        // Just go on unwinding.  
        trace!("unwinding: skipping frame  
            self.pop_stack_frame(/* unwinding  
            return Ok(true);  
    }  
};  
let basic_block = 8self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx(  
        self.statements(stmt)?;  
    return Ok(true);  
}  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
create fn statement(8mut self, stmt: 8mir::Statem  
    info!("{:?}", stmt);  
    use rustc_middle::mir::Statem  
    // See
```

Why port Rust to Morello?

- The guarantees of capabilities complement the guarantees of Rust

```
    &mut self) → InterpResult<'tcx, ...> {
        self.step()? {}

        ns: `true` as long as there are more...
        is used by [priroda](https://github.com/priroda/priroda)

        is marked `#inline(always)` to work...
        e(always)]
        step(&mut self) → InterpResult<'tcx, ...> {
            self.stack().is_empty() {
                return Ok(false);
            }

            let loc = match self.frame().loc {
                Ok(loc) ⇒ loc,
                Err(_) ⇒ {
                    // We are unwinding and this fn.h
                    // Just go on unwinding.
                    trace!("unwinding: skipping frame");
                    self.pop_stack_frame(/* unwinding */);
                    return Ok(true);
                }
            };
        };
        let basic_block = &self.body().basic_block;
        let old_frames = self.frame_idx();
        if let Some(stmt) = basic_block.statements
            .get(old_frames, self.frame_idx()) {
            self.statement(stmt)?;
            return Ok(true);
        }

        M::before_terminator(self)?;

        let terminator = basic_block.terminator();
        assert_eq!(old_frames, self.frame_idx());
        self.terminator(terminator)?;
        Ok(true)
    }

    /// Runs the interpretation logic for the given
    /// statement counter. This also moves the state
    /// info!("{:?}", stmt);
    use rustc_middle::mir::Statement;
    // See
```


Why port Rust to Morello?

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

```
    &mut self) → InterpResult<'tcx>
    self.step()? {}

    ns: 'true' as long as there are more
    is used by [priroda](https://github.com/priroda)

    is marked '#inline(always)' to work
    e(always)]
    step(&mut self) → InterpResult<'tcx>
    self.stack().is_empty() {
    return Ok(false);

    let loc = match self.frame().loc {
    .. Ok(loc) => loc,
    .. Err(_) => {
    // We are unwinding and this fn has
    // Just go on unwinding.
    trace!("unwinding: skipping frame");
    self.pop_stack_frame(/* unwinding
    return Ok(true);
    };
    let basic_block = &self.body().basic_block;
    let old_frames = self.frame_idx();
    if let Some(stmt) = basic_block.statements
    assert_eq!(old_frames, self.frame_idx());
    self.statements(stmt)?;
    return Ok(true);
    }

    M::before_terminator(self)?;

    let terminator = basic_block.terminator();
    assert_eq!(old_frames, self.frame_idx());
    self.terminator(terminator)?;
    Ok(true)
    }

    /// Runs the interpretation logic for the given
    /// statement counter. This also moves the state
    crate fn statement(&mut self, stmt: &mir::Statement,
    info!("[{:?}]", stmt);

    use rustc_middle::mir::TerminatorKind;
    // See
```

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

```
8mut self) → InterpResult<'tcx  
self.step()? {}  
  
ns `true` as long as there are mor  
is used by [priroda](https://github  
is marked `#inline(always)` to wor  
e(always)]  
step(&mut self) → InterpResult<'tcx  
self.stack().is_empty() {  
return Ok(false);  
  
et loc = match self.frame().loc {  
Ok(loc) ⇒ loc,  
Err(_) ⇒ {  
    // We are unwinding and this fn h  
    // Just go on unwinding.  
    trace!("unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding  
    return Ok(true);  
};  
let basic_block = &self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx(  
    self.statements(stmt)?;  
    return Ok(true);  
}  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(&mut self, stmt: &mir::Statem  
    info!("{:?}", stmt);  
    use rustc_middle::mir::Statem  
    // See
```

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

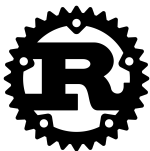


Digital Security
by Design

```
8mut self) → InterpResult<'tcx  
self.step()? {}  
  
ns `true` as long as there are mor  
is used by [priroda](https://github  
is marked `#inline(always)` to wor  
e(always)]  
step(&mut self) → InterpResult<'tcx  
self.stack().is_empty() {  
return Ok(false);  
  
et loc = match self.frame().loc {  
Ok(loc) ⇒ loc,  
Err(_) ⇒ {  
    // We are unwinding and this fn h  
    // Just go on unwinding.  
    trace!("unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding  
    return Ok(true);  
};  
let basic_block = &self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
assert_eq!(old_frames, self.frame_idx(  
self.statements(stmt)?;  
return Ok(true);  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx(  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(&mut self, stmt: &mir::Statement  
info!("{:?}", stmt);  
use rustc_middle::mir::S  
// See
```

Why port Rust to Morello?

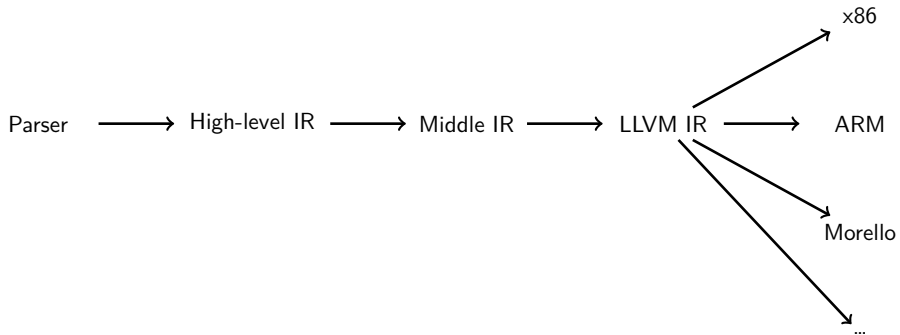
- ▶ The guarantees of capabilities complement the guarantees of Rust
- ▶ Rust provides compile-time guarantees for safe code
- ▶ Capabilities provide run-time guarantees for **unsafe** code



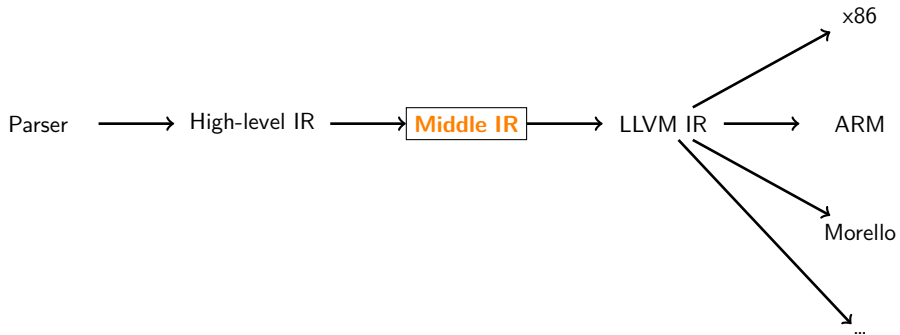
Digital Security
by Design

```
8mut self) → InterpResult<'tcx  
self.step()? {}  
  
ns `true` as long as there are mor  
is used by [priroda](https://github  
is marked `#inline(always)` to wor  
e(always)]  
step(&mut self) → InterpResult<'tcx  
self.stack().is_empty() {  
return Ok(false);  
  
et loc = match self.frame().loc {  
Ok(loc) ⇒ loc,  
Err(_) ⇒ {  
    // We are unwinding and this fn h  
    // Just go on unwinding.  
    trace!("unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding  
    return Ok(true);  
};  
let basic_block = &self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
assert_eq!(old_frames, self.frame_idx(  
self.statements(stmt)?;  
return Ok(true);  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(&mut self, stmt: &mir::Statement  
info!("{:?}", stmt);  
use rustc_middle::mir::S  
// See
```

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...

```
fn step(&mut self) → InterpreterResult<'tcx, ...> {
    self.step()? { }

    ...
    ns: 'true' as long as there are more
    is used by [priroda](https://github.com/
    ...
    is marked '#inline(always)' to work
    e(always)]
    step(&mut self) → InterpreterResult<'tcx, ...> {
    self.stack().is_empty() {
    return Ok(false);

    let loc = match self.frame().loc {
    ... Ok(loc) ⇒ loc,
    ... Err(_) ⇒ {
    // We are unwinding and this fn has
    // Just go on unwinding.
    trace!("unwinding: skipping frame");
    self.pop_stack_frame(/* unwinding
    return Ok(true); } Mark Roussk

    };
    let basic_block = &self.body().basic_block;
    let old_frames = self.frame_idx();

    if let Some(stmt) = basic_block.statements
    assert_eq!(old_frames, self.frame_idx(
    self.statements(stmt)?;
    return Ok(true);

    M::before_terminator(self)?;

    let terminator = basic_block.terminator();
    assert_eq!(old_frames, self.frame_idx());
    self.terminator(terminator)?;
    Ok(true)

    /// Runs the interpretation logic for the given
    /// statement counter. This also moves the state
    crate fn statement(&mut self, stmt: &mir::Statement,
    info!("[{:?}]", stmt);

    use rustc_middle::mir::Statement;
    // See
```

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.

```
fn mut_step(&mut self) → InterpreterResult<'tcx> {
    self.step()? {}

    // ... true as long as there are more ...
    // ... is used by [priroda](https://github.com/...
    // ... is marked #inline(always) to work ...
    // ... (always)]
    step(&mut self) → InterpreterResult<'tcx> {
        self.stack().is_empty() {
            return Ok(false);
        }

        let loc = match self.frame().loc {
            Ok(loc) ⇒ loc,
            Err(_) ⇒ {
                // We are unwinding and this fn ...
                // Just go on unwinding.
                trace!("unwinding: skipping frame ...
                self.pop_stack_frame(/* unwinding ...
                return Ok(true);
            }
        };
        let basic_block = &self.body().basic_block ...
        let old_frames = self.frame_idx();
        if let Some(stmt) = basic_block.statements ...
            assert_eq!(old_frames, self.frame_idx( ...
            self.statements(stmt)?;
            return Ok(true);
        }

        M::before_terminator(self)?;
        let terminator = basic_block.terminator();
        assert_eq!(old_frames, self.frame_idx());
        self.terminator(terminator)?;
        Ok(true)
    }

    /// Runs the interpretation logic for the given ...
    /// statement counter. This also moves the state ...
    crate fn statement(&mut self, stmt: &mir::Statement ...
        info!("{:?}", stmt);
        use rustc_middle::mir::Statement ...
        // See ...
```


Compiler changes — object layout

```
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()
        },
    }
}
```

```
6mut self) -> InterpResult<tcx  
self.step()? { }
```

```
ns `true` as long as there are mo
```

```
is used by [priroda](https://github
```

```
is marked `#inline(always)` to wor
```

```
e(always)]
```

```
step(&mut self) -> InterpResult<tcx
```

```
self.stack().is_empty() {
```

```
return Ok(false);
```

```
et loc = match self.frame().loc {
```

```
Ok(loc) => loc,
```

```
Err(_) => {
```

```
// We are unwinding and this fn h
```

```
// Just go on unwinding.
```

```
trace!("unwinding: skipping frame
```

```
self.pop_stack_frame(// unwinding
```

```
return Ok(true);
```

```
};
```

```
let basic_block = &self.body().basic_block
```

```
let old_frames = self.frame_idx();
```

```
if let Some(stmt) = basic_block.statements
```

```
assert_eq!(old_frames, self.frame_idx(
```

```
self.statements(stmt)?;
```

```
return Ok(true);
```

```
};
```

```
/// before_terminator(self)?;
```

```
let terminator = basic_block.terminator();
```

```
assert_eq!(old_frames, self.frame_idx(
```

```
self.terminator(terminator)?;
```

```
Ok(true)
```

```
};
```

```
/// Runs the interpretation logic for the given
```

```
/// statement counter. This also moves the state
```

```
crate fn statement(&mut self, stmt: &mir::Statem
```

```
info!("{:?}", stmt);
```

```
use rustc_middle::mir::Statem
```

```
/// See
```

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.

```
8mut self) → InterpResult<'tcx  
self.step()? {}  
  
ns `true` as long as there are mor  
is used by [priroda](https://github  
is marked `#inline(always)` to wor  
e(always)]  
step(&mut self) → InterpResult<'tcx  
self.stack().is_empty().{  
return Ok(false);  
  
let loc = match self.frame().loc {  
Ok(loc) ⇒ loc,  
Err(_) ⇒ {  
    // We are unwinding and this fn h  
    // Just go on unwinding.  
    trace!("unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding  
    return Ok(true);  
};  
let basic_block = &self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx(  
    self.statements(stmt)?;  
    return Ok(true);  
}  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
create fn statement(&mut self, stmt: &mir::Statement  
    info!("{:?}", stmt);  
use rustc_middle::mir::S  
// See
```

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());  
}
```

```
    &mut self) → InterpreterResult<'tcx,  
    self.step()? {}  
  
    ns: 'true' as long as there are more  
    is used by [priroda](https://github.com/priroda/priroda)  
    is marked '#inline(always)' to work  
    e(always)]  
    step(&mut self) → InterpreterResult<'tcx,  
    self.stack().is_empty() {  
    return Ok(false);  
  
    let loc = match self.frame().loc {  
    Ok(loc) ⇒ loc,  
    Err(_) ⇒ {  
        // We are unwinding and this fn has  
        // just go on unwinding.  
        trace!("unwinding: skipping frame");  
        self.pop_stack_frame(/* unwinding */);  
        return Ok(true);  
    }  
    };  
    let basic_block = &self.body().basic_block;  
    let old_frames = self.frame_idx();  
    if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx());  
    self.statements(stmt);  
    return Ok(true);  
}  
  
M::before_terminator(self);  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator);  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(&mut self, stmt: &mir::Statement,  
    info!("[{:?}]", stmt);  
    use rustc_middle::mir::StatementKind;  
    // See
```

Standard library changes

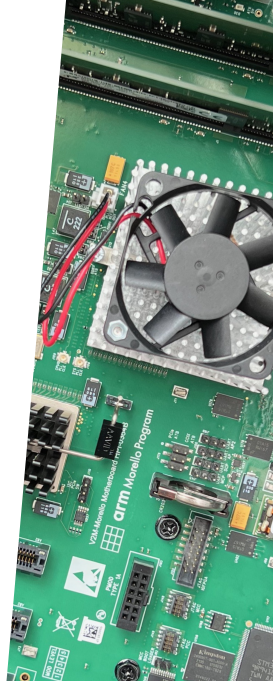
- ▶ We're not done here, yet.
- ▶ The worst so far has been in a concurrency library which casts pointers to/from integers to tag them with metadata in the lower bits.
- ▶ Some bits of the FFI needed some tweaks, integer types being replaced with pointer types.

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

```
    &mut self) -> InterpResult<'tcx  
    self.step()? {}  
  
    ns: 'true' as long as there are mor  
    is used by [priroda](https://githu  
    is marked '#inline(always)' to wor  
    e(always)]  
    step(&mut self) -> InterpResult<'tcx  
    self.stack().is_empty() {  
    return Ok(false);  
  
    let loc = match self.frame().loc {  
    Ok(loc) => loc,  
    Err(_) => {  
        // We are unwinding and this fn h  
        // Just go on unwinding.  
        trace!("unwinding: skipping frame  
        self.pop_stack_frame(/* unwinding  
        return Ok(true);  
    }  
};  
let basic_block = &self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx(  
        self.statements(stmt)?;  
    return Ok(true);  
}  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(&mut self, stmt: &mir::Statem  
    info!("{:?}", stmt);  
use rustc_middle::mir::Statem  
    // See
```

What's done

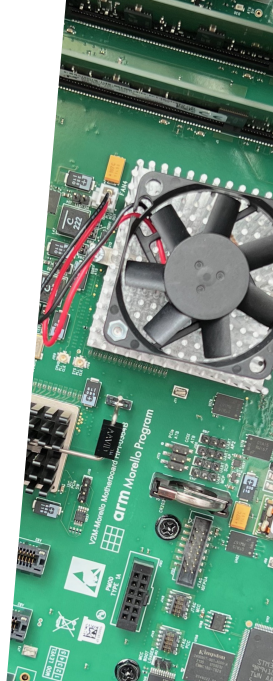
- ▶ The compiler emits Purecap code for the Morello machine.
- ▶ The core part of the standard library is ported, and fairly well tested.
- ▶ Various parts of the Rust infrastructure are ported to Morello.
- ▶ `std` is ported, but not as thoroughly tested and there are some known bugs to work through.



What's done

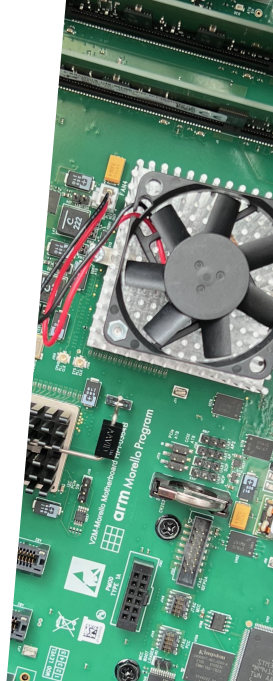
- ▶ The compiler emits Purecap code for the Morello machine.
- ▶ The core part of the standard library is ported, and fairly well tested.
- ▶ Various parts of the Rust infrastructure are ported to Morello.
- ▶ `std` is ported, but not as thoroughly tested and there are some known bugs to work through.

Demo



Remaining challenges

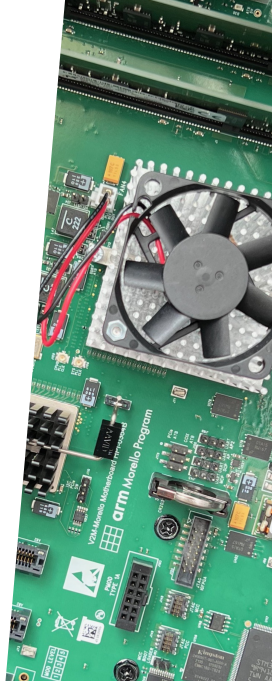
- ▶ Porting the rest of the standard library.
- ▶ Looking at some code-gen bugs which appear to come from LLVM.
- ▶ Port substantial 3rd party software built in Rust to Morello.
- ▶ Two optimisations are disabled: SROA, and Function Merging.



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)

$$\llbracket P \rrbracket_{\text{MIR}}$$

```
    &mut self) → InterpreterResult<tcx, Self> {
        self.step()? {}

        // is marked `true` as long as there are more frames
        // is used by [priroda](https://github.com/priroda/priroda)
        // is marked `#inline(always)` to work around a bug in rustc
        // (always)
        step(&mut self) → InterpreterResult<tcx, Self> {
            self.stack().is_empty() {
                return Ok(false);
            }

            let loc = match self.frame().loc {
                Ok(loc) ⇒ loc,
                Err(_) ⇒ {
                    // We are unwinding and this fn.h
                    // Just go on unwinding.
                    trace!("unwinding: skipping frame");
                    self.pop_stack_frame(/* unwinding */);
                    return Ok(true);
                }
            };
            // Mark Roussk

            let basic_block = &self.body().basic_block;
            let old_frames = self.frame_idx();

            if let Some(stmt) = basic_block.statements
            .get(old_frames) {
                self.statements(stmt)?;
                return Ok(true);
            }

            M::before_terminator(self)?;

            let terminator = basic_block.terminator();
            assert_eq!(old_frames, self.frame_idx());
            self.terminator(terminator)?;
            return Ok(true);
        }

        /// Runs the interpretation logic for the given
        /// statement counter. This also moves the state
        /// counter.
        fn statement(&mut self, stmt: &mir::Statement) {
            info!("{:?}", stmt);
            use rustc_middle::mir::StatementKind;
            // See
        }
    }
}
```

Formal Verification Goals

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

$$\llbracket \text{opt}(P) \rrbracket_{\text{MIR}} \subseteq \llbracket P \rrbracket_{\text{MIR}}$$

```
    &mut self) → InterpreterResult<tcx, ...  
    self.step()? {}  
  
    ns `true` as long as there are more  
    is used by [priroda](https://github.com/priroda/priroda)  
    is marked `#inline(always)` to work  
    e(always)]  
    step(&mut self) → InterpreterResult<tcx,  
    self.stack().is_empty() {  
    return Ok(false);  
  
    let loc = match self.frame().loc {  
    ... Ok(loc) => loc,  
    ... Err(_) => {  
    // We are unwinding and this fn has  
    // Just go on unwinding.  
    trace!("unwinding: skipping frame",  
    self.pop_stack_frame(/* unwinding  
    return Ok(true);  
    }  
    };  
    let basic_block = &self.body().basic_block;  
    let old_frames = self.frame_idx();  
    if let Some(stmt) = basic_block.statements  
    ... assert_eq!(old_frames, self.frame_idx());  
    ... self.statements(stmt)?;  
    ... return Ok(true);  
    }  
    M::before_terminator(self)?;  
    let terminator = basic_block.terminator();  
    assert_eq!(old_frames, self.frame_idx());  
    self.terminator(terminator)?;  
    Ok(true)  
    }  
  
    /// Runs the interpretation logic for the given  
    /// statement counter. This also moves the state  
    crate fn statement(&mut self, stmt: &mir::Statement,  
    info!("[{:?}]", stmt);  
    use rustc_middle::mir::interpret::InterpCx;  
    // See
```

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

$$\llbracket \text{rustc}(P) \rrbracket_{\text{MIR}} \subseteq \llbracket P \rrbracket_{\text{RUST}}$$

```
fn mut_self() → InterpreterResult<'tcx> {
    self.step()? {}

    // ... true as long as there are more ...
    // ... is used by [priroda](https://github.com/priroda/priroda) ...
    // ... is marked `#inline(always)` to work around a bug in rustc ...
    // ... (always) ...
    step(&mut self) → InterpreterResult<'tcx> {
        self.stack().is_empty() {
            return Ok(false);
        }

        let loc = match self.frame().loc {
            Ok(loc) ⇒ loc,
            Err(_) ⇒ {
                // We are unwinding and this fn has no loc.
                // Just go on unwinding.
                trace!("unwinding: skipping frame");
                self.pop_stack_frame(/* unwinding */);
                return Ok(true);
            }
        };
        // Mark Roussigne

        let basic_block = &self.body().basic_block;
        let old_frames = self.frame_idx();

        if let Some(stmt) = basic_block.statements.get(old_frames) {
            assert_eq!(old_frames, self.frame_idx());
            self.statements(stmt)?;
            return Ok(true);
        }

        M::before_terminator(self)?;

        let terminator = basic_block.terminator();
        assert_eq!(old_frames, self.frame_idx());
        self.terminator(terminator)?;
        return Ok(true);
    }

    /// Runs the interpretation logic for the given statement.
    /// This also moves the state counter.
    fn statement(&mut self, stmt: &mir::Statement) {
        info!("{:?}", stmt);
        use rustc_middle::mir::StatementKind;
        // ...
    }
}
```

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

$$\llbracket \text{rustc}(P) \rrbracket_{\text{CHERI}} \subseteq \llbracket P \rrbracket_{\text{MIR}}$$

```
    &mut self) → InterpreterResult<'tcx, ...  
    self.step()? {}  
  
    ns: 'true' as long as there are mor...  
    is used by [priroda](https://github...  
    is marked '#inline(always)' to wor...  
    e(always)]  
    step(&mut self) → InterpreterResult<'tcx,  
    self.stack().is_empty() {  
    return Ok(false);  
  
    let loc = match self.frame().loc {  
    Ok(loc) ⇒ loc,  
    Err(_) ⇒ {  
        // We are unwinding and this fn h...  
        // Just go on unwinding.  
        trace!("unwinding: skipping frame...  
        self.pop_stack_frame(/* unwinding...  
        return Ok(true);  
    };  
    let basic_block = &self.body().basic_block...  
    let old_frames = self.frame_idx();  
    if let Some(stmt) = basic_block.statements...  
    assert_eq!(old_frames, self.frame_idx(...  
    self.statements(stmt)?;  
    return Ok(true);  
}  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given...  
/// statement counter. This also moves the state...  
create fn statement(&mut self, stmt: &mir::Statem...  
    info!("{:?}", stmt);  
    use rustc_middle::mir::Statem...  
    // See
```

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

$$\text{safe}(P) \implies \nexists X \in \llbracket \text{rustc}(P) \rrbracket_{\text{CHERI}}. \text{faulty}(X)$$

```
fn mut_self) → InterpResult<tcx  
self.step()? {}  
  
ns `true` as long as there are mor  
is used by [priroda](https://github  
is marked `#inline(always)` to wor  
e(always)]  
step(&mut self) → InterpResult<tcx  
self.stack().is_empty() {  
return Ok(false);  
  
let loc = match self.frame().loc {  
Ok(loc) ⇒ loc,  
Err(_) ⇒ {  
    // We are unwinding and this fn h  
    // Just go on unwinding.  
    trace!("unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding  
    return Ok(true);  
};  
let basic_block = &self.body().basic_block  
let old_frames = self.frame_idx();  
if let Some(stmt) = basic_block.statements  
    assert_eq!(old_frames, self.frame_idx(  
        self.statements(stmt)?;  
    return Ok(true);  
}  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frame_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
crate fn statement(&mut self, stmt: &mir::Statement  
    info!("{:?}", stmt);  
use rustc_middle::mir::S  
// See
```

Thanks for listening!
Any questions?

