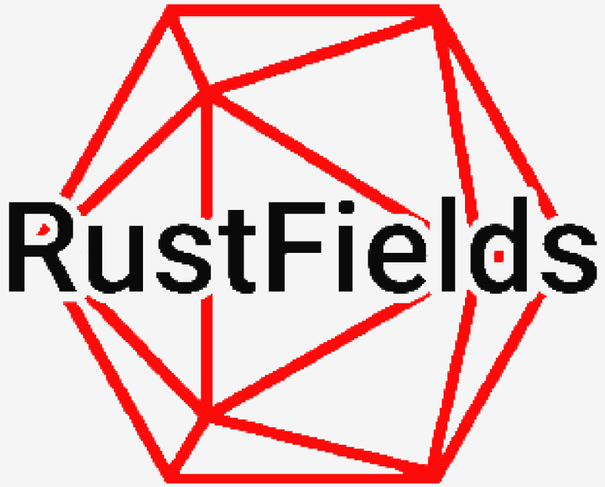


Angela Cortecchia – Leonardo Micelli – Paolo Penazzi – Filippo Vissani

Pervasive Computing – @UniBo

Introduction & Goals

This project is born from the willingness to extend ScaFi: a Scala-based library and framework for Aggregate Programming.

After a meeting with the stakeholders, the following goals have been identified:

- Update the original ScaFi core to Scala 3
- Create a Rust version of ScaFi
- Find a way to make the two versions interoperable
- Expand the test suite
- Standardize the message format

Design ideas

Given the exploratory nature of the project, the team has considered changing some aspects of the Scafi architecture:

- Making the VM immutable
- Making fields first-class entities
- Introducing other constructs of aggregate programming such as Exchange and Share

Exploration Outcomes (1/2)

It has been decided to maintain the original architecture to ensure compatibility and interoperability.

Additionally, a change in architecture would have taken time away from integrating with Rust, which is the core and most complex part of the project.

In the end:

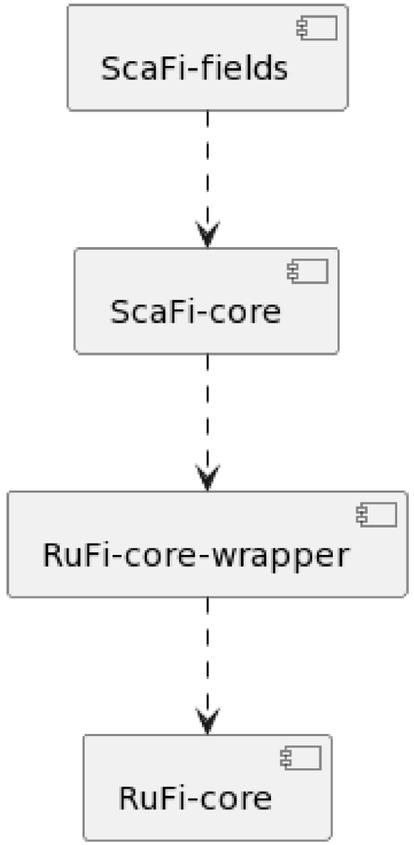
- The Scala3 and Rust versions of ScaFi maintain the original architecture.
- The use of reified fields is enabled by an external library for ScaFi.

Exploration Outcomes (2/2)

Regarding the Scala-Rust integration, we found two distinct solutions, that both involve the linking of a Shared Library with Scala Native:

- Write an integration layer between RuFi-core and Scala Native
- Rewrite RuFi-core to be C-interoperable

Overarching Architecture



On ScaFi Fields

- Developed on top of the ScaFi-core
- We designed a Field[A] type and re-implemented the core constructs so that they utilize it
- We enriched the Field type with a Monad instance so that it could be used inside comprehensions

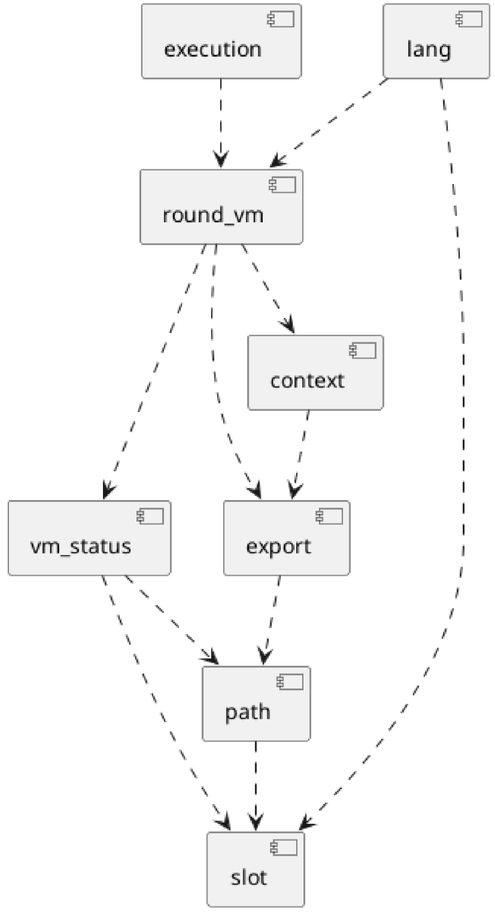
```
class Example extends FieldLanguageProgram:
  override type MainResult = Int
  new *
  override def main() =
    val f1 = nbrf(mid())
    val f2 = Field.lift(2)
    val f3 = for
      id <- f1
      n <- f2
    yield id + n
    foldhoodf[Int](_+_)(f3)
```

On ScaFi

The work done on ScaFi Core was essentially updating the library from Scala 2 to Scala 3, without significant refinements.

On RuFi

We decided to stick to the current ScaFi-core architecture as much as possible even for the Rust counterpart, because reckoned it would be beneficial for maintenance and interoperability purposes.



Rust Issues (1/3)

In Rust, it is not possible to have both a mutable and an immutable borrow of the same variable within the same scope simultaneously.

This is an issue because language constructs can’t be implemented like they are in Scala.

We evaluated different solutions like Cells and Dipendency Injection but both didn’t solve our problem.

We ended up changing the signature of the language construct to take the VM as a parameter and return it along the expression result.

```
pub fn rep<A: Copy + 'static, F, G>(mut vm: RoundVM, init: F, fun: G) -> (RoundVM, A)
```

Rust Issues (2/3)

As a result of the changes made to the language constructs' signatures, we were compelled to change the type of the expr parameter in the methods.

Typically, within the expression, there is another nested construct, which will need to take the VM as a parameter and return it along the expression result.

```
pub fn rep<A: Copy + 'static, F, G>(mut vm: RoundVM, init: F, fun: G) -> (RoundVM, A)
    where
    F: Fn() -> A,
    G: Fn(RoundVM, A) -> (RoundVM, A),
{
```

Rust Issues (3/3)

The foldhood construct resulted as problematic due to the following facts:

- The folded_eval code couldn’t be written inside the foldhood as it is in Scala due to borrowing rules. We decided to implement an auxiliary recursive function that computes the folded eval for each aligned neighbour

Examples – Rust

```rust
// Program: rep(9)(_ * 2)
let program : fn(RoundVM) -> (...) = |vm : RoundVM| rep(vm,
                                init: || 9,
                                fun: |vm1 : RoundVM, a : i32| (vm1, a * 2));
```

```rust
// Program: foldhood(-5)(_ + _)(nbr(2))
let program : fn(RoundVM) -> (...) = |vm : RoundVM| foldhood(vm,
                                init: || -5,
                                aggr: | a : i32, b : i32| (a + b),
                                expr: |vm1 : RoundVM| nbr( vm: vm1,  expr: |vm2 : RoundVM| (vm2, 2)));
```

Interoperability Issues (1/4)

```
#[no_mangle]
pub extern "C" fn local_sense<A: 'static>(&self, sensor_id: &SensorId) → Option<&A> {
    self.context.local_sense::<A>(sensor_id)
}

```

Warning: functions generic over types or consts must be mangled

Interoperability Issues (2/4)

```
#[no_mangle]
pub extern "C" fn new(context: Context) → Self {
    Self {
        vm: RoundVM::new_empty(context)
    }
}
```

Warning: ` extern` fn uses type ` Context` , which is not FFI-safe

Interoperability Issues (3/4)

```
1 #[no_mangle]
2 pub extern "C" fn new(context: Context) → Self {}
```

Error: symbol `new` is already defined

Can be fixed using the export_name attribute

Interoperability Issues (4/4)

`struct { int x, y; }*`	`unsafe.Ptr[unsafe.CStruct2[unsafe.CInt, unsafe.CInt]]` [2] [5]
`struct { int x, y; }`	Not supported

Scala Native has limited compatibility with C data structures

Future Works

- Complete the integration layer between Scala and Rust
- Find a way to standardize the messages exchanged by devices in the network
- Explore a solution that makes use of Exchange or Share instead of the current constructs
- Expand the test suite of each project
- Expand the Rust core

Thanks for the attention!