# Using Functional Reactive Programming to Define Safe Actor Systems

Nick Webster
Victoria University of Wellington
Wellington, New Zealand
nick.webster@vuw.ac.nz

Marco Servetto
Victoria University of Wellington
Wellington, New Zealand
marco.servetto@ecs.vuw.ac.nz

Michael Homer
Victoria University of Wellington
Wellington, New Zealand
michael.homer@ecs.vuw.ac.nz

## ABSTRACT

Functional Reactive Programming (FRP) is a powerful abstraction for building deterministic concurrent systems. However, some programmers prefer a more imperative approach for certain tasks, and that approach is required to implement some imperative algorithms. The Actor Model provides an abstraction for building concurrent systems in a more imperative way without as much of the chaos typical of traditional shared-memory imperative concurrent programming. While the Actor Model offers more structure than other imperative approaches, it still suffers from nondeterminism due to message-ordering and processing times. That makes actor systems hard to reason about, limiting their effectiveness for critical tasks. We formally define an elegant multi-paradigm unification of event-driven FRP constructs and the Actor Model. Our unification enables an intuitive form of declarative programming that can integrate imperative and declarative code within each other. We use reference and object capabilities to tame imperative features: reference capabilities track aliasing and mutability, and object capabilities track I/O. Notably, in our system expressions with deeply immutable input behave deterministically. Additionally, capabilities provide a boundary to allow nondeterministic code to intermingle safely with deterministic code.

## CCS CONCEPTS

• **Computing methodologies** → **Concurrent programming languages**; **Parallel programming languages**; • **Theory of computation** → *Concurrency*.

## KEYWORDS

Functional programming, Functional reactive programming, Actor model, Concurrency, Declarative programming, Type systems

## 1 INTRODUCTION

Parallel programming promises great performance improvements, but it is also a source of undesired nondeterministic behaviour. *Functional Reactive Programming* (FRP) and the *Actor Model* offer two different ways to tame nondeterminism in concurrent and/or distributed systems.


*FTfJP '22, June 7, 2022, Berlin, Germany*


In the actor model, each actor sees the world sequentially, and processes a single message at a time [1]. Actors may perform computations, I/O, mutate their own private state, and send messages to other actors when they process a message. However, messages can be delivered in an unpredictable order and because actors can mutate their private state, they can behave nondeterministically based on message delivery ordering.

FRP comes in many flavours but the common element is that each approach provides a deterministic way to work with values that change with respect to time [3]. The core concept in *Event-Driven FRP* (E-FRP) is called a *signal*, with the definition: `Signal` $\alpha \approx$ `Time ->` $\alpha$[11]. With E-FRP the `Time` input can be misleading to think about because it does not refer to any real monotonic clock; `Time` is defined as progressing every time an event occurs. An E-FRP program generally consists of a series of *signal functions* that accept a `Signal a` and transform it into a `Signal b`. Signal functions can deterministically run in parallel (assuming glitch freedom) because immutability shields them from observing any parallelism. A key issue with FRP systems is a lack of expressivity for working with the "awkward squad" of tasks, like I/O and concurrency [12]. To resolve that issue, some E-FRP languages have to rely on an external environment written in a different language to perform side effects and provide nondeterministic inputs [14].

### 1.1 FRJ

If we think of each event in a signal as a series of messages and each signal function as an actor's message handler, we can represent an E-FRP program deterministically with the Actor Model. Conversely, if each message was a signal that only has one event and every actor's message handler was a signal function, we can represent a pure actor system with an E-FRP program. The duality between E-FRP and the Actor Model is the foundation of *Featherweight Reactive Java* (FRJ), our object calculus that unifies E-FRP constructs with the Actor Model to enable the definition of safe actor systems.

Events and messages are identical in FRJ, so the term "event" from E-FRP and the term "message" from the Actor Model can be used interchangeably when talking about FRJ's signals and actors. FRJ's signals are applicative functors represented by a linked data-structure: `@[`$e$`;`$e'$`]`. The head of this data-structure is an expression that evaluates to the current value of the signal, and the tail is an expression that will evaluate to a new linked data-structure with a head expression that evaluates to the next value of the signal, and so on. The laziness or strictness of the evaluation of these expressions isn't defined by our formalism. The only requirements on the evaluation of these expressions is that the runtime must finish evaluating the head expression before starting to evaluate the tail and that the evaluation happens asynchronously. Our signal's internal data

structure can be introspected using the conventional `head(_)` and `tail(_)` syntax. However, users will rarely directly interact with this abstraction and will mostly use lifted method calls. Most useful abstractions that rely on directly using `head`, `tail`, and the underlying representation of FRJ's signals can be encapsulated into the body of generic library methods. Lifted method calls have a special syntax: `a.@m(b,c)`, where `b` and `c` are signals. This syntax sends the actor `a` a message causing the (asynchronous) computation of `a.m(head(b),head(c))` and then triggers `a.@m(tail(b),tail(c))`; until either `b` or `c` terminates.

Consider the following class:

```
capability class FuelSensor { Int resAddrP; Int resAddrF;
  mut method Num pressure() =
    new PciBus(this.resAddrP).etc(...);
  mut method Num fuel() =
    new PciBus(this.resAddrF).etc(...);
}
class Format {
  method String of(Num f, Num p) = f+"% at "+p+"Pa"; }
```

Given a `FuelSensor` object (`fs`), we could write the conventional method call `new Format().of(fs.fuel(),fs.pressure())`, to compute the status string once. Using FRJ's lifted method calls, we can write the following in an FRP style:

```
@Num pressures = new FuelSensor(0x12,0x34).@pressure();
new Format().of(new FuelSensor(0x12,0x34).@fuel(),pressures);
```

That creates a signal with messages containing the formatted status string. As the readings on fuel and pressure change, the formatted string will change too. We can also write code in an actor style, by sending individual events to `FuelSensor`. This code sends the messages, `fuel` and `pressure`, to the `FuelSensor` actors one time. The two `FuelSensor` actors will reply with a `Num` message each. When both messages are handled, `Format` receives a single message asking to produce the formatted string, parameterised over the fuel percentage and the pressure. The expressions inside signals are computed in parallel and execution is deferred. Thus, implementing a fork–join is trivial in FRJ:

```
@Int part1=@[x.computePart1();@[]],
Int part2=x.computePart2(),
head(part1)+part2;
```

The fork-join works because the creation of `part1` does not block because its head is being evaluated in parallel. The `head(part1)` call would block until the expression had been computed and the message was ready.

With actor frameworks like Akka [8], a distinction is made between methods in an actor object and message handlers. However, because FRJ supports lifting of methods into signal functions, and because of our unification of actors and FRP, a lifted method can be seen as both a signal function *and* a message handler. Therefore, any object that can have a method lifted is also an actor in FRJ.

## 2 FORMAL MODEL

The values ($v$) in FRJ consist of memory addresses ($L$), signal references ($S$), and evaluated messages. Our memory is modelled as a series of $\rho$, which are maps from memory addresses to objects and their mailboxes. All messages ($\boxtimes$) are tagged with a signal

$$
\begin{array}{lcl}
cap & ::= & \texttt{capability} \mid \emptyset \\
CD & ::= & cap\ \ \texttt{class}\ C\ \texttt{implements}\ \overline{C}\ \{\ \overline{F}\ \overline{M};\ \} \\
 & & \mid cap\ \texttt{interface}\ C\ \texttt{extends}\ \overline{C}\ \{\ MH_1; \ldots MH_n;\ \} \\
F & ::= & T\ f; \\
T & ::= & mdf\ C \mid \texttt{@}T \\
MH & ::= & mdf\ \texttt{method}\ T\ m\ (T_1\ x_1 \ldots T_n\ x_n) \\
M & ::= & MH = e; \\
e & ::= & x \mid e.m(\overline{e}) \mid e.f \mid e_1.f{=}e_2 \mid \texttt{new}\ C(\overline{e}) \mid T\ x = e_1,\ e_2 \\
 & & \mid e.\texttt{@}m(\overline{e}) \mid \texttt{@[}e;e'\texttt{]} \mid \texttt{@[]} \mid \texttt{head}(e) \mid \texttt{tail}(e) \\
v & ::= & L \mid S \mid [v;\ S] \\
\mathcal{E} & ::= & \square \mid \mathcal{E}.m(\overline{e}) \mid v.m(\overline{v}\ \mathcal{E}\ \overline{e}) \mid \mathcal{E}.f \mid \mathcal{E}.f{=}e \mid v.f{=}\mathcal{E} \\
 & & \mid \texttt{new}\ C(\overline{v}\ \mathcal{E}\ \overline{e}) \mid T\ x = \mathcal{E}, e \mid \mathcal{E}.\texttt{@}m(\overline{e}) \\
 & & \mid v.\texttt{@}m(\overline{v}\ \mathcal{E}\ \overline{e}) \mid \texttt{head}(\mathcal{E}) \mid \texttt{tail}(\mathcal{E}) \\
mdf & ::= & \texttt{imm} \mid \texttt{mut} \mid \texttt{capsule} \mid \texttt{read} \\
\boxtimes & ::= & S[e_1;e_2] \\
\mu & ::= & \rho_1 \ldots \rho_n \\
\rho & ::= & L \mapsto C(\overline{v})\ \overline{\boxtimes} \\
\Gamma & ::= & x_1 : T_1 \ldots x_n : T_n \\
\Sigma & ::= & L_1 : C_1 \ldots L_n : C_n
\end{array}
$$

**Figure 1: The grammar of FRJ**

reference. The shape of the reduction is: $\mu \mid e \rightarrow \mu' \mid e'$. We use $class(C)$ to denote the class declaration ($CD$) for the class $C$ and $fields(C)$ to denote the list of the fields for the class $C$. Additionally, '_' is used as a placeholder in the rules and can match any syntactic term. Free metavariables are fresh.

### 2.1 Well Formedness

Using the auxiliary notation, our well-formedness rules are as follows:

- All classes and interfaces are uniquely named.
- All methods in a given class are uniquely named.
- All fields in a given class are uniquely named.
- All parameters in a given method are uniquely named and are not called `this`.
- A `capsule` binding can be used zero or one time in an expression.
- All $S$ labelling a $\boxtimes$ inside the memory are unique.
- Fields can only have the type modifiers: `imm` or `mut`.
- Types containing `@` must have the `imm` modifier.
- Classes can only implement interfaces.
- Interfaces can only extend other interfaces.
- `@[]` is not in $domS(\mu)$ (defined below).
- $\mu \mid e$ is well formed if all $L$ in $e$ are in $dom(\mu)$ (defined below) and all $usedS(e) \cup usedS(\mu)$ are in $domS(\mu)$.

$dom(\mu)$ is the conventionally defined set of all keys ($L$) in the memory ($\mu$). $domS(\mu)$ is the set of all $S$ labelling a $\boxtimes$ inside the memory, and $usedS(\mu)$ (i.e. all signals reachable or executing) is defined as follows:

- $usedS(\mu) = usedS(\mu, \rho_1) \cup \ldots \cup usedS(\mu, \rho_n)$, with $\mu = \rho_1 \ldots \rho_n$
- $usedS(L \mapsto C(v_1 \ldots v_k)\ \ \boxtimes_1 \ldots \boxtimes_n) = usedS(v_1) \cup \ldots \cup usedS(v_k) \cup usedS(\boxtimes_1) \cup \ldots \cup usedS(\boxtimes_n)$
- $S \in usedS(S'[e_1;e_2]) \ = usedS(e_1) \cup usedS(e_2)$ if $S \neq S'$

- $S \in usedS(e)$ if $S$ is a sub-expression of $e$.

## 2.2 Reduction Rules

$$\frac{L \mapsto C(v_1 \ldots v_n) \;\; \_ \text{ in } \mu \quad T_1\, f_1 \ldots T_n\, f_n = fields(C)}{\mu \mid L.f_i \rightarrow \mu \mid v_i}(\text{fAccess})$$

$$\frac{\begin{array}{c}\_\; T_0\, f_0 \ldots T_n\, f_n = fields(C) \\ \rho_0 = L \mapsto C\ (\overline{v}\, v_0 \ldots v_n)\ \overline{\boxtimes} \\ \rho_1 = L \mapsto C\ (\overline{v}\, v\, v_1 \ldots v_n)\ \overline{\boxtimes}\end{array}}{\mu, \rho_0 \mid L.f_0 = v \rightarrow \mu, \rho_1 \mid v}(\text{fUpdate})$$

$$\frac{}{\mu \mid \texttt{new}\ C(v_1 \ldots v_n) \rightarrow \mu, L \mapsto C(v_1 \ldots v_n)\ \emptyset \mid L}(\text{new})$$

$$\frac{\begin{array}{c}L \mapsto C(\overline{v})\ \overline{\boxtimes} \text{ in } \mu \\ \_\ \texttt{method}\ \_\ m\ (\_\ x_1 \ldots \_\ x_n) = e; \text{ in } class(C)\end{array}}{\begin{array}{c}\mu \mid L.m(v_1 \ldots v_n) \rightarrow \\ \mu \mid e[\texttt{this} = L, x_1 = v_1 \ldots x_n = v_n]\end{array}}(\text{mCall})$$

$$\frac{}{\mu \mid T\ x{=}v, e \rightarrow \mu \mid e[x = v]}(\text{let})$$

$$\frac{\mu \mid e \rightarrow \mu \mid e'}{\mu \mid \mathcal{E}[e] \rightarrow \mu \mid \mathcal{E}[e']}(\mathcal{E})$$

$$\frac{\mu \mid e \rightarrow \mu \mid e'}{\mu,\ \rho\, S\texttt{[}\mathcal{E}[e]\texttt{;}e_0\texttt{]}\ \mid e_1 \rightarrow \mu,\ \rho\, S\texttt{[}\mathcal{E}[e']\texttt{;}e_0\texttt{]}\ \mid e_1}(\mathcal{E}\text{Head})$$

$$\frac{\mu \mid e \rightarrow \mu \mid e'}{\mu,\ \rho\, S\texttt{[}v\texttt{;}\mathcal{E}[e]\texttt{]}\ \mid e_1 \rightarrow \mu,\ \rho\, S\texttt{[}v\texttt{;}\mathcal{E}[e']\texttt{]}\ \mid e_1}(\mathcal{E}\text{Tail})$$

Field updates, field access, object construction and method call are standard. All objects in FRJ hold an initially empty mailbox for messages. Contextual rules ($\mathcal{E}$, $\mathcal{E}$Head, and $\mathcal{E}$Tail) guide the parallel reduction: ($\mathcal{E}$) allows us to reduce the main expression, while ($\mathcal{E}$Head) reduces the current value of a signal. When the value is produced, rule ($\mathcal{E}$Tail) executes the expression creating the next signal node. Memory ($\mu$) is a set, so the rules can work on any $\rho$ in $\mu$. The $\rho$ non-terminal has a list of ✉ at its end; thus by writing $\mu,\ \rho\, S\texttt{[}e\texttt{;}e'\texttt{]}$ we are selecting the last message of an arbitrary object in memory.

$$\frac{}{\mu \mid \texttt{head([}v\texttt{;}S\texttt{])} \rightarrow \mu \mid v}(\text{head}) \qquad \frac{}{\mu \mid \texttt{tail([}v\texttt{;}S\texttt{])} \rightarrow \mu \mid S}(\text{tail})$$

$$\frac{}{\mu \mid \texttt{tail(@[ ])} \rightarrow \mu \mid \texttt{@[ ]}}(\text{tailEmpty})$$

$$\frac{\begin{array}{c}\text{either } e = \mathcal{E}[\texttt{head(@[ ])}] \\ \text{or } e = v \text{ and } e_0 = \mathcal{E}[\texttt{head(@[ ])}]\end{array}}{\mu,\ \rho\, S\texttt{[}e\texttt{;}e_0\texttt{]}\ \mid e_1 \rightarrow (\mu,\ \rho\ \mid e_1)[S = \texttt{@[ ]}]}(\text{empty})$$

$$\frac{}{\mu,\ \rho\, S\texttt{[}v\texttt{;}S'\texttt{]}\ \mid e_1 \rightarrow (\mu,\ \rho\ \mid e_1)[S = \texttt{[}v\texttt{;}S'\texttt{]}]}(\text{complete})$$

When a message has been completely computed, rule (complete) removes the message from the memory, and replaces all of the references to $S$ with $\texttt{[}v\texttt{;}S'\texttt{]}$. So, while $\texttt{head}(S)$ will cause the reduction to get stuck, $\texttt{head([}v\texttt{;}S\texttt{])}$ can reduce. Therefore, the rule (complete) enables a form of synchronisation between the messages and their consumers.

When the message execution tries to access the head of the empty signal, rule (empty) terminates the signal, removes the message $S$ and replaces all occurrences of $S$ with `@[ ]`. This rule also allows the computation of a signal to be cancelled deterministically by explicitly using `head(@[])`.

$$\frac{}{\mu \mid \texttt{@[}e_1\texttt{;}e_2\texttt{]} \rightarrow \mu, L \mapsto \texttt{Object()}\ S\texttt{[}e_1\texttt{;}e_2\texttt{]} \mid S}(\text{explicitS})$$

$$\frac{\begin{array}{c}e_0 = L.\texttt{@}m(v_1 \ldots v_n) \\ e_1 = L.m(\texttt{head}(v_1) \ldots \texttt{head}(v_n)) \\ e_2 = L.\texttt{@}m(\texttt{tail}(v_1) \ldots \texttt{tail}(v_n))\end{array}}{\mu, L \mapsto C(\overline{v})\ \overline{\boxtimes} \mid e_0 \rightarrow \mu, L \mapsto C(\overline{v})\ S\texttt{[}e_1\texttt{;}e_2\texttt{]}\ \overline{\boxtimes} \mid S}(\text{liftS})$$

This group of rules (explicitS and liftS) deals with the creation of signals.

For the explicit creation of signals, the rule (explicitS) reduces signal constructors into a message (✉) and places it on a new empty actor. The signal constructor expression is then replaced with the fresh signal ($S$) that was just associated with the message.

The alternative way to create signals in FRJ is through lifting methods. Rule (liftS) reduces lifted method calls by creating a ✉ that gets placed onto the receiver containing a head of the traditional method call with arguments of the head of all of its inputs. The tail of this new ✉ will be the same lifted method call, but with the tail of all of the inputs as the inputs for the new lifted call. Effectively, the method now *reacts* to its inputs.

$$\frac{}{\mu, \mu' \mid e \rightarrow \mu \mid e}(\text{garbage})$$

Rule (garbage) gets rid of the part of memory that is unreachable starting from the main expression. Note that we cannot arbitrarily split the memory. We can only split it in such a way that the resulting $\mu \mid e$ is well formed. An important consequence of our garbage collection rule is that messages can be collected too, even during their computation. However, due to our well-formedness rules, messages can only be collected if the receiver actor object is collected, and an object can only be collected if there are no other references to its address and to any of the $S$ in its mailbox.

## 2.3 Reference and Object Capabilities

Parallel computation is inherently part of FRP and actor systems. FRJ uses reference capabilities to tame the nondeterminism that would otherwise arise from aliasing and mutability. FRJ supports the three most common reference capabilities: the default `imm` (deeply immutable); `mut` (mutable) and `read` (readonly) the common supertype of both `imm` and `mut`. In addition, FRJ supports `capsule`; a reference that dominates its $\text{ROG}_{\texttt{mut}}$ (mutable reachable object graph) [6]. For each `mut` or `capsule` reference we can compute

$\text{ROG}_{\texttt{mut}}(L) = \overline{L}$ by following all the `mut` fields. Formally:

$L \in \text{ROG}_{\texttt{mut}}(L, \mu)$

$L_i \in \text{ROG}_{\texttt{mut}}(L, \mu)$ if $L \mapsto C(L_1 \ldots L_n), \_ \in \mu,\ fields(C) = T_1 \ldots T_n \wedge T_i = \texttt{mut}\ \_$

$L_2 \in \text{ROG}_{\texttt{mut}}(L_0, \mu)$ if $L_1 \in \text{ROG}_{\texttt{mut}}(L_0, \mu) \wedge L_2 \in \text{ROG}_{\texttt{mut}}(L_1, \mu)$

$\text{ROG}_{\texttt{mut}}(\_, \_)$ is only defined for `mut` and `capsule` references; it is undefined for `read` and `imm` ones.

All references without an explicit capability are `imm`. Recalling the `FuelSensor` class from earlier, we saw the use of two different reference capabilities: `imm`, `mut`. If we look at the pressure method: `mut method Num pressure() = new PciBus(this.resAddrP).etc(..)`

The receiver (`this`) is of type `mut FuelSensor` reference and the return value will be an `imm Num` reference. This means that to call this method on an object, a `mut FuelSensor` reference is required.

A `capsule` reference is the sole access point to a group of mutable objects. `capsule` references can be obtained when the aliasing is under control. The whole $\text{ROG}_{\texttt{mut}}$ from a capsule reference can only be reached from that specific capsule reference. Note that fields can only hold `imm` and `mut` references. Reference capabilities have the following subtyping relationship:

`read` $\leq$ `_` $\leq$ `capsule`

All of the reference capabilities are subtypes of `capsule` and supertypes of `read`. Thanks to this a `capsule` reference can be converted to an `imm` or a `mut` reference. Note that `mut` and `imm` are not comparable with each other.

For example, we can extend the `FuelSensor` class from earlier with the following methods:

```
1 capability class FuelSensor { /* ... */
2   capsule method @Num throttle(@Bool tick) =
3     this.@_throttle(tick);
4   mut method Num _throttle(Bool tick) = this.fuel();
5 }
```

The `throttle` method synchronises reading from the sensor with a clock signal, enabling us to keep readings from all sensors in sync and throttle the signal throughput. Line 3 lifts the `_throttle` method. `throttle` is a capsule method because the type system requires lifted methods on capability objects to have `imm` or `capsule` receivers. Line 3 is very interesting. The "tick" from the clock is discarded but the argument is still important because the method is now reacting to the clock, synchronising with it.

The main advantage of reference capabilities over older forms of aliasing control [2, 7], is that references can be promoted/recovered to a subtype when the right conditions arise. *Multiple method types* offers one way to handle promotion/recovery. In the case of FRJ every method has two additional types, one replacing `mut` with `capsule` and another replacing `mut` with `capsule` and `read` with `imm`:

$$
\begin{aligned}
&methTypes(T_0, m) = \{ \\
&\quad T_0 \ldots T_n \mapsto T, \\
&\quad (T_0 \ldots T_n \mapsto T)[\texttt{mut} = \texttt{capsule}], \\
&\quad (T_0 \ldots T_n \mapsto T)[\texttt{mut} = \texttt{capsule}, \texttt{read} = \texttt{imm}] \\
&\} \text{ iff} \\
&\quad mdf\ \texttt{method}\ T\ m\ (T_1\ x_1 \ldots T_n\ x_n)\ \_ \in class(C) \\
&\quad T_0 = mdf'\ C \qquad mdf' \leq mdf
\end{aligned}
$$

Where the notation [`mut = capsule`], replaces all of the `mut` modifiers with `capsule`.

For example, consider the following method signature, that could be a getter for a `mut List` field: `read method read List getList();` The following additional method type will be available in the type system: `method List getList();`, enabling an `imm List` to be returned if the receiver is `imm`.

Given the following method signature, that could be applying an in-place transformation algorithm on a list:
`method mut List transform(mut List list, read Algo algo)`
The following additional method type will be available in the type system: `method capsule List transform(capsule List list, read Algo algo)`. In this way simple `mut->mut` methods can transparently work on `capsule`s. Note how `read` parameters are also transparently allowed.

Many languages allow any piece of code to do I/O, typically by indirectly calling native static methods. If static methods are forbidden, we can get a more pure OO setting where all behaviour is obtained by method calls to objects. Put simply, if there's no object to do the task, the task can't be done. This principle is the key idea behind object capabilities [5, 10]. One approach to object capabilities is having a privileged standard library that can freely perform I/O and has a security model for restricting access to its privileged methods. FRJ places the restrictions at the language level instead of the possibly-buggy library level. Some classes are labelled as `capability`, and only privileged expressions (`mut` methods in capability classes/the main method) can instantiate a capability class. Our reduction rules do not model I/O interaction but in a realistic implementation of FRJ we would expect a way to run native code in `mut` methods of capability classes. In this way, any method that only has immutable references for parameters is guaranteed to be deterministic. We can be sure of the guarantee because:

(1) New `mut` objects can be created anywhere, but new `mut capability` objects can only be created in the `main` expression or `mut` methods in `capability` classes.
(2) We do not allow any static variables, so you can only access objects through explicit channels like the parameters and the receiver.
(3) The ROG of an `imm` reference does not contain any non-`imm` references, so `mut` capability objects cannot be accessed.

## 2.4 Type Rules

FRJ's typing environment has three components: $\Gamma$, the mapping between variables and types; $\Sigma$, the mapping between a object locations and class names; and *cap*, a flag identifying if the expression is allowed to instantiate capability classes.

We will use the notation $capOf(C)$ and $capOf(T)$ to denote the capability modifier of a given class. For simplicity's sake, the *class*, *fields*, and *capOf* helper functions can also take a $T$ value as an argument and consider the $C$ inside of it.

$$\frac{}{cap; \Sigma; \Gamma \vdash x : \Gamma(x)}(x) \qquad \frac{cap; \Sigma; \Gamma \vdash e : T' \quad \vdash T' \leq T}{cap; \Sigma; \Gamma \vdash e : T}(\text{sub})$$

$$\frac{}{cap; \Sigma; \Gamma \vdash L : mdf\ \Sigma(L)}(L)$$

Variable typing and subsumption are standard. We omit the subtyping judgement, which would be standard but with the addition

of signal types; those would require the level of indirection to be the same, i.e. $@T' \leq @T$ if $T' \leq T$ but $@@T' \not\leq @T$.

The rule (L) types memory references as the class of the object it points to and the modifier of the reference. Our rules are very declarative; a more explicit approach could instrument the expressions to keep track of the pair $L{:}mdf$ during reduction. This could be needed to complete a proof of soundness. At compile-time the actual memory addresses wouldn't be known but the mapping between a $L$ placeholder and its class would be.

$$\frac{cap;\Sigma;\Gamma \vdash e : mdf\ C \quad fields(C) = T_1\ f_1 \ldots T_n\ f_n}{cap;\Sigma;\Gamma \vdash e.f_i : T_i \triangleright mdf}(\text{fAccess})$$

$$\frac{cap;\Sigma;\Gamma \vdash e_1 : \texttt{mut}\ C \quad fields(C) = T_1\ f_1 \ldots T_n\ f_n \quad cap;\Sigma;\Gamma \vdash e_2 : T_i}{cap;\Sigma;\Gamma \vdash e_1.f_i{=}e_2 : T_i}(\text{fUpdate})$$

Field access and field update are conventional with the exception of modifiers being applied to the result of a field access, and the added requirement that the receiver of a field update must be `mut`. FRJ's field access modifier composition rules are as follows:

- $\overline{@}\ mdf\ C \triangleright \texttt{imm} = \overline{@}\ \texttt{imm}\ C$
- $\overline{@}\ mdf\ C \triangleright \texttt{mut} = \overline{@}\ mdf\ C$
- $\overline{@}\ mdf\ C \triangleright \texttt{capsule} = \overline{@}\ mdf\ C$
- $\overline{@}\ \texttt{mut}\ C \triangleright \texttt{read} = \overline{@}\ \texttt{read}\ C$
- $\overline{@}\ \texttt{imm}\ C \triangleright \texttt{read} = \overline{@}\ \texttt{imm}\ C$

For example, with a field access, if the receiver had the `read` modifier and the field had the `imm` modifier, the result would be `imm`. Alternatively, if the receiver was `read` and the field was `mut`, the result would be `read`.

$$\frac{T_1\ f_1 \ldots T_n\ f_n = fields(C) \quad cap;\Sigma;\Gamma \vdash e_i : T_i \quad \text{either } capOf(C) = \emptyset \text{ or } cap = \texttt{capability}}{cap;\Sigma;\Gamma \vdash \texttt{new}\ C(e_1 \ldots e_n)\ :\ \texttt{mut}\ C}(\text{new})$$

Object instantiation is also mostly conventional, defining the type of the expression as a `mut` reference to the object being instantiated. The major difference is that if the class is marked as `capability`, then the object can only be created in the main method or in a `mut` method of another capability class. The rules for capability methods can be found in the rule (method).

$$\frac{T_1\ f_1 \ldots T_n\ f_n = fields(C) \quad cap;\Sigma;\Gamma \vdash e_i : T_i[\texttt{mut} = mdf] \quad \text{either } capOf(C) = \emptyset \text{ or } cap = \texttt{capability} \quad mdf = \{\texttt{imm}, \texttt{capsule}\}}{cap;\Sigma;\Gamma \vdash \texttt{new}\ C(e_1 \ldots e_n)\ :\ mdf\ C}(\text{new+})$$

The rule (new+) enables a more flexible creation of objects. They can be `imm` if all the fields are initialised with `imm`s/`capsule`s, or can be `capsule` if all the `mut` fields are initialised with `capsule`s. This is safe because fields can only be `imm` or `mut`. `imm` fields can remain unchanged and if all `mut` fields are encapsulated, the newly created object and its reachable object graph is encapsulated. This is not the only way to create capsules, multiple method types can also be used: every method that returns a `mut` but does not receive any `mut` arguments implicitly returns a `capsule` too.

$$\frac{cap;\Sigma;\Gamma \vdash e_0 : T_0 \quad cap;\Sigma;\Gamma, x : T_0 \vdash e_1 : T_1}{cap;\Sigma;\Gamma \vdash T_0\ x{=}e_0,\ e_1 : T_1}(\text{let})$$

The let typing rule is conventional.

$$\frac{T_0 \ldots T_n \mapsto T \text{ in } methTypes(T_0, m) \quad cap;\Sigma;\Gamma \vdash e_i : T_i}{cap;\Sigma;\Gamma \vdash e_0\ .m(e_1 \ldots e_n) : T}(\text{mCall})$$

$$\frac{cap;\Sigma;\Gamma \vdash e_0 : T_0 \quad cap;\Sigma;\Gamma \vdash e_i : @T_i \quad \forall i \in 1..n \quad T_0 \ldots T_n \mapsto T \text{ in } methTypes(T_0, m) \quad validActor(T_0)}{cap;\Sigma;\Gamma \vdash e_0\ .@m(e_1 \ldots e_n) : @T}(\text{mCall@})$$

Our method call type rule is mostly conventional but relies on $methTypes$, and thus is more flexible. The major difference between rule (mCall) and rule (mCall@) is that all of the argument types are lifted ($@T$) and the receiver must be a $validActor$: either the receiver is immutable ($T_0 = \texttt{imm}\ \_$) or the receiver is a deeply encapsulated capability instance ($capOf(T_0) = \texttt{capability}$ and $T_0 = \texttt{capsule}\ \_$).

Actors may receive messages in any order; while immutable actors cannot be influenced by such order, a mutable actor (even if it is fully encapsulated) may use the messages to update the value of a field. If such an actor could be freely created, then we could use it to forge a no-args method with a nondeterministic result.

$$\frac{cap;\Sigma;\Gamma[\text{only } \texttt{imm},\ \texttt{capsule}] \vdash e_1 : T \quad cap;\Sigma;\Gamma[\text{only } \texttt{imm},\ \texttt{capsule}] \vdash e_2 : @T}{cap;\Sigma;\Gamma \vdash @[e_1;e_2] : @T}(\text{fullSignal})$$

The rule (fullSignal) is for a signal constructor.
The notation $\Gamma[\text{only } \texttt{imm}, \texttt{capsule}] = \Gamma'$ creates a new $\Gamma'$ environment with only `imm` and `capsule` bindings. This enforces that only `imm` and `capsule` variables can be captured by the expressions inside the signal.

$$\frac{}{cap;\Sigma;\Gamma \vdash @[\,] : @T}(\text{emptySignal})$$

The rule (emptySignal) is similar to the conventional rule for typing empty lists, as the empty signal can assume any signal type, similar to how `null` is valid for any boxed type in Java.

$$\frac{cap;\Sigma;\Gamma \vdash e : @T}{cap;\Sigma;\Gamma \vdash \texttt{head}(e) : T}(\text{head}) \qquad \frac{cap;\Sigma;\Gamma \vdash e : @T}{cap;\Sigma;\Gamma \vdash \texttt{tail}(e) : @T}(\text{tail})$$

$$\frac{overrideOk(C', MH_i) \quad \forall C' \in \overline{C} \quad \text{either } cap = \texttt{capability} \text{ or } capOf(C') = \emptyset \quad \forall C' \in \overline{C}}{\vdash cap\ \texttt{interface}\ C\ \texttt{extends}\ \overline{C}\ \{MH_1 \ldots MH_n\}\ \text{OK}}(\text{interface})$$

$$\frac{cap;C \vdash M_i \quad overrideOk(C', M_i) \quad \forall C' \in \overline{C} \quad dom(C') \subseteq dom(C) \quad \forall C' \in \overline{C} \quad \text{either } cap = \texttt{capability} \text{ or } capOf(C') = \emptyset \quad \forall C' \in \overline{C}}{\vdash cap\ \ \texttt{class}\ C\ \texttt{implements}\ \overline{C}\ \{\ \overline{F}\ K\ M_1 \ldots M_n\ \}\ \text{OK}}(\text{class})$$

$$\frac{cap';\emptyset;\ \texttt{this} : mdf\ C, x_1 : T_1 \ldots x_n : T_n \vdash e : T \quad cap' = \emptyset \text{ iff } cap = \emptyset \text{ or } mdf \neq \texttt{mut}}{cap, C \vdash mdf\ \texttt{method}\ T\ m\ (T_1\ x_1 \ldots T_n\ x_n) = e;}(\text{method})$$

Those last three rules type interfaces, classes, and methods. In particular, rule (method) types all `mut` methods in `capability` classes as capability methods. We omit the trivial but tedious definition for $overrideOk(C', MH_i)$, checking if a method signature can override a potential method with the same name defined in the super interface: if another method with the same name exists, the two method types must be identical.

## 3 EXAMPLE

We discussed the capability class `FuelSensor` earlier, which directly connects to the hardware (using `PciBus`) and lifts the hardware sensor to a signal with the method `throttle`, allowing us to synchronise the sensor input across any number of signals using the same clock.

The produced signal from `throttle` is of type `@Num`; having no connection with `FuelSensor` type or any capability restriction. Thus, we can use deterministic and pure code like `Smooth` to manipulate it.

```
class Smooth { method @Num of(@Num ns) =
  new Math().@average(ns, tail(ns)); }
//Using Smooth to reduce noise on a single sensor
@Bool t = new Std().clock(50), /* clock cycle */
capsule FuelSensor o2 = new FuelSensor(0x1234, 0x12),
@Num res = new Smooth().of(o2.throttle(t)),
```

Relying on a conventional `Num Math.average(Num,Num)` function, `Smooth.of` smooths out the noise from the signal: the current value of the signal is going to be the average of the former value and the current one. The iconic FRP function `foldp` (fold over the past) [4] can similarly be implemented by explicitly using `tail(_)`.

In the same way we can smooth the signal coming from a single sensor, we can take a consensus vote from the signals of multiple hardware sensors in order to make our system more reliable in case of hardware failure.

```
@Bool t = new Std().clock(50), /* clock cycle */
@Num o2s1 = new Smooth().of(
  new FuelSensor(0x12, 0x34).throttle(t)),
@Num o2s2 = new Smooth().of(
  new FuelSensor(0x56, 0x78).throttle(t)),
@Num o2s3 = new Smooth().of(
  new FuelSensor(0x91, 0x92).throttle(t)),
@Number o2 = new MajorityVote().@of(o2s1,o2s2,o2s3),
```

The following class `LaunchControl` shows a useful programming pattern: an object where all of its fields are signals can combine those signals into a single one simply by providing a method combining the individual signal values into the result.

```
class LaunchControl { @Bool approved;
  @Num o2; /* Oxygen% */ @Num rp1; /* fuel% */
  method @Bool canLaunch() = this.@_canLaunch(this.approved,
 this.o2, this.rp1);
  method Bool _canLaunch(Bool approved, Num o2, Num rp1) =
    approved && o2 == 100 && rp1 == 100; }
```

With all of our inputs configured, we can now launch our rocket!

```
capability interface Engine {
  capsule method Status igniteWhen(Bool shouldIgnite); }
class Rocket { mut Engine engines; LaunchControl control;
  capsule method @Status launch() = this.engines.@igniteWhen
(this.control.canLaunch()); }
main = /*...*/, new Rocket(/*..*/).launch();
```

## 4 RELATED WORK

This work largely builds on top of an honours report on unifying actors with FRP [17], which additionally features a prototype compiler for a more primitive version of the formal model described in this paper. Additionally, 42 [6, 13] is a strong inspiration for this work and shares a similar kind of multi-method promotion approach, reference capabilities, and uses object capabilities for controlling I/O. However, 42 doesn't use actors or signal functions as its main abstraction to support parallelism.

XFRP [14] offers an interesting model for executing pure FRP on an actor-based runtime. Using XFRP would be a similar experience to using FRJ without object capabilities and with every reference being `imm`. The language has fewer sources of nondeterminism to control because it delegates side effects to components that are external to the program.

In 2019 Lohstroh et al. [9] proposed a new actor system that uses *reactors*. Reactors declare their inputs and outputs, react to messages, and are connected with a 'composite' main function that builds the graph. The 'reactor network' can offer stronger guarantees for message delivery/processing order than traditional shared-memory actor libraries like Akka [8]. However, the system cannot enforce properties on behaviour like the absence of data races without enforcing strict ordering requirements.

The 'actor-reactor model' (ARM) [16] replaces actors with reactors and creates a joint model where actors can be nondeterministic. The reactors in the ARM should not be confused with the Lohstroh et al.'s reactors; ARM's reactors are always pure and deterministic. Notably the ARM's reactors are what the authors call "strongly reactive", which means that they are limited to performing $O(1)$ operations. FRJ does not have the distinction between 'actors' and 'reactors'. FRJ's unified approach does still make a distinction between deterministic and nondeterministic actors using object capabilities, but a pure FRJ actor is able to perform more complex tasks than an ARM reactor because we do not require signal functions/message handlers to be strongly reactive.

Pony is a programming language for creating shared-memory actor systems. Pony offers reference and object capabilities to provide guarantees for their shared-memory actor model implementation [15]. Pony's reference capabilities mostly map to the ones used in FRJ and references can only be sent to other actors when it would be indistinguishable from deep-copying them into a message. Unlike FRJ, Pony's object capability system is implemented at the library level instead of at the language level with `capability` classes.

## REFERENCES


[1] Gul A Agha. 1985. *Actors: A model of concurrent computation in distributed systems.* Technical Report. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab.

[2] John Boyland. 2003. Checking interference with fractional permissions. In *International Static Analysis Symposium.* Springer, 55–72.

[3] Evan Czaplicki. 2012. Elm: Concurrent FRP for functional guis. *Senior thesis, Harvard University* (2012).

[4] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. *ACM SIGPLAN Notices* 48, 6 (2013), 411–422.
[5] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. 2008. Verifiable functional purity in Java. In *Proceedings of the 15th ACM conference on Computer and communications security*. 161–174.
[6] Paola Giannini, Marco Servetto, Elena Zucca, and James Cone. 2019. Flexible recovery of uniqueness and immutability. *Theoretical Computer Science* 764 (2019), 145–172.
[7] John Hogg. 1991. Islands: Aliasing protection in object-oriented languages. In *Conference proceedings on Object-oriented programming systems, languages, and applications*. 271–285.
[8] Lightbend Inc. [n.d.]. *Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala.* Retrieved 2021-12-30 from https://akka.io/
[9] M. Lohstroh and E. A. Lee. 2019. Deterministic Actors. In *2019 Forum for Specification and Design Languages (FDL)*. 1–8.
[10] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-Based Module System for Authority Control. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 20:1–20:27. https://doi.org/10.4230/LIPIcs.ECOOP.2017.20
[11] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. *SIGPLAN Not.* 51, 12 (sep 2016), 33–44. https://doi.org/10.1145/3241625.2976010
[12] Simon Peyton Jones. 2001. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.* IOS Press, 47–96. https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/
[13] Marco Servetto. [n.d.]. *42 - Metaprogramming as default.* Retrieved 2022-04-13 from https://l42.is/
[14] Kazuhiro Shibanai and Takuo Watanabe. 2018. Distributed functional reactive programming on actor-based runtime. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 13–22.
[15] George Steed and Sophia Drossopoulou. 2016. A principled design of capabilities in Pony. *Master's thesis, Imperial College* (2016).
[16] Sam Van den Vonder, Joeri De Koster, Florian Myter, and Wolfgang De Meuter. 2017. Tackling the awkward squad for reactive programming: the actor-reactor model. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. 27–33.
[17] Nick Webster. 2020. Using Functional Reactive Programming to define Actor Systems.