Fault Tolerant Functional Reactive Programming (Functional Pearl)

IVAN PEREZ, National Institute of Aerospace, USA

Highly critical application domains, like medicine and aerospace, require the use of strict design, implementation and validation techniques. Functional languages have been used in these domains to develop synchronous dataflow programming languages for reactive systems. Causal stream functions and Functional Reactive Programming capture the essence of those languages in a way that is both elegant and robust.

To guarantee that critical systems can operate under high stress over long periods of time, these applications require clear specifications of possible faults and hazards, and how they are being handled. Modeling failure is straightforward in functional languages, and many Functional Reactive abstractions incorporate support for failure or termination. However, handling *unknown types of faults*, and incorporating *fault tolerance* into Functional Reactive Programming, requires a different construction and remains an open problem.

This work presents extensions to an existing functional reactive abstraction to facilitate tagging reactive transformations with hazard tags or confidence levels. We present a prototype framework to quantify the reliability of a reactive construction, by means of numeric factors or probability distributions, and demonstrate how to aid the design of fault-tolerant systems, by constraining the allowed reliability to required boundaries. By applying type-level programming, we show that it is possible to improve static analysis and have compile-time guarantees of key aspects of fault tolerance. Our approach is powerful enough to be used in systems with realistic complexity, and flexible enough to be used to guide their analysis and design, to test system properties, to verify fault tolerance properties, to perform runtime monitoring, to implement fault tolerance during execution and to address faults during runtime. We present implementations in Haskell and in Idris.

CCS Concepts: • **Software and its engineering → Functional languages**; • **Hardware → Fault tolerance**; • **Theory of computation →** *Streaming models*;

Additional Key Words and Phrases: fault tolerance, functional reactive programming, stream programming, streams, reactive programming, monads, monad transformers, dependent types

ACM Reference Format:
Ivan Perez. 2018. Fault Tolerant Functional Reactive Programming (Functional Pearl). *Proc. ACM Program. Lang.* 2, ICFP, Article 96 (September 2018), 30 pages. https://doi.org/10.1145/3236791

1 INTRODUCTION

Mission critical systems – those in which a malfunction may result in loss of life or great economic impact – require the use of careful design, implementation, testing and deployment techniques. In domains like aviation and transportation, the development of both hardware and software is heavily regulated and strict guidelines must be followed.

The use of synchronous programming languages with clear semantics is frequent in these domains [Berry et al. 2000; Dormoy 2008; Halbwachs et al. 1991]. These languages are normally

Author's address: Ivan Perez, National Institute of Aerospace, USA, ivan.perez@nianet.org.

2475-1421/2018/9-ART96
https://doi.org/10.1145/3236791

based on a notion of streams and stream functions, and they must guarantee, at compile time, that all streams are causal and well-formed.

The essence of these languages can be captured via causal stream functions and some forms of Functional Reactive Programming (FRP) [Courtney et al. 2003; Elliott and Hudak 1997; Nilsson et al. 2002]. While FRP has traditionally been applied to domains like interactive programming and games, abstractions like Monadic Stream Functions [Perez 2017; Perez et al. 2016] help bridge the gap between discrete-time causal stream programming and continuous-time FRP.

However, even when functional reactive and synchronous dataflow systems are implemented according to specifications, they may fail in production due to undetected software bugs in other components, hardware failures, or environmental hazards. It is extremely important to determine the most likely hardware and software faults and environmental damage, and to minimize their impact by introducing mechanisms for *fault tolerance* [Avižienis 1967, 1976; Butler 2008].

A general way of capturing and addressing faults in typed functional languages is by means of optional values or values that encode errors, and possibly the reasons behind those errors. FRP frameworks like Monadic Stream Functions and Yampa introduce notions of failure or termination via the use of *Maybe* and *Either*.

While these types capture notions of *detectable failures*, they are not suitable to represent *failures that are not detectable* or *correctable* at a given stage. For example, in space applications, we normally see bit flips due to radiation, which alter values in memory. Without further adjustments to our program, we may not be able to tell that an error occurred, let alone correct it.

This paper presents mechanisms to encode, at the type level, the possibility of undetectable failures in a functional specification of a reactive system. We do so in three ways: by tagging values with different degrees of confidence, by means of probability distributions denoting their expected failure rates, and by tagging values with value-level and type-level fault sets that denote the kinds of failures that may have affected them.

This extra information serves two purposes: At compile time, it helps guide system design, giving us clear type-level specification of the kinds of faults that have not been corrected at a particular stage, or the trust that we can place on that design. At runtime, it helps understand the confidence we can place on a specific result, and make dynamic adjustments and decisions about a mission based on the margins of error we can tolerate. We present two implementations, in Haskell and in Idris, based on an extension of Monadic Stream Functions to work with parameterized monads, and demonstrate which kinds of analysis are possible in each language.

Our work makes the following contributions:

- We extend Monadic Stream Functions, an existing formalism for Functional Reactive Programming that combines Arrows with Monads, with support for *parameterized* or *indexed monads*, which facilitates chaining constructions working on different monads.
- We present a polymorphic type constructor to encode *potentially unreliable data*, and show how it can be used to compute the expected reliability or availability of reactive computations.
- We provide a representation of certainty based on *probability* distributions, and show how to calculate the total confidence of a reactive network.
- We show how to use *type-level programming* to capture *potential faults* at the type level and delimit fault tolerant areas, and show how to leverage fault analysis on the type system to obtain a proof of the faults that may affect the behavior of a reactive network.
- We demonstrate how to combine unreliable data to implement *fault tolerance* mechanisms like voting in ways that improve the reliability of the original data sources.

The rest of this paper is structured as follows. Section 2 introduces basic concepts in FRP and fault tolerance, and illustrates the problem we seek to address. Section 3 introduces a type to

tag values with reliability factors, and shows how to write monadic computations that combine potentially unreliable data from multiple sources. Section 4 shows how handle unreliable data and increase its confidence by introducing *fault tolerance*. Section 5 extends the previous approach by assigning probability distributions to values. Section 6 presents how to represent fault sets, and uses the same approach as before to include tag reactive results with the fault sets that may have affected them. Section 7 introduces Monadic Stream Functions over parameterized monads. Section 8 presents how to represent fault sets at the type level, and uses type-level programming to statically specify potential faults and obtain evidence of the faults that have and have not been handled by an operation. Section 9 presents two implementations of our work, in Haskell and in Idris, and compares their expressiveness. Section 10 presents related work. Section 11 presents our conclusions and future work.

2 BACKGROUND

In the interest of making this paper sufficiently self-contained, we summarize the basics of FRP and Monadic Stream Functions in the following. We later describe the basic ideas from the field of fault tolerance, together with an example that illustrates the problems we seek to address. For further details on FRP and Arrowized FRP (AFRP), see earlier papers [Courtney et al. 2003; Elliott and Hudak 1997; Nilsson et al. 2002]. This presentation draws heavily from the summaries in [Courtney et al. 2003; Perez and Nilsson 2017]. For details on fault tolerance, see [Avižienis 1967, 1976] and, for an in-depth introduction, see [Butler 2008].

2.1 Functional Reactive Programming

FRP is a programming paradigm to describe hybrid systems that operate on time-varying data. FRP is structured around the concept of *signal*, which conceptually can be seen as a function from time to values of some type:

$$Signal\ a \approx Time \longrightarrow a$$

Time is (notionally) continuous, and is represented as a non-negative real number. The type parameter *a* specifies the type of values carried by the signal. For example, the type of an animation would be *Signal Picture* for some type *Picture* representing static pictures. Signals can also represent input data, like the mouse position.

Additional constraints are required to make this abstraction executable. First, it is necessary to limit how much of the history of a signal can be examined, to avoid memory leaks. Second, if we are interested in running signals in real time, we require them to be *causal*: they cannot depend on other signals at future times. FRP implementations address these concerns by limiting the ability to sample signals at arbitrary points in time.

The space of FRP frameworks can be subdivided into two main branches, namely Classic FRP [Elliott and Hudak 1997] and Arrowized FRP [Nilsson et al. 2002]. Classic FRP programs are structured around signals or a similar notion representing internal and external time-varying data. In contrast, Arrowized FRP programs are defined using causal functions between signals, or *signal functions*, connected to the outside world only at the top level. While FRP is conceptually continuous, implementations still execute by sampling inputs at discrete points in time, and some even hide time completely.

To address some of the limitations of different forms of FRP, and bridge the gap between Classic and Arrowized FRP, and between continuous-time and discrete-time variants, Monadic Stream Functions (MSFs) separate the notion of time from the notion of causal transformation over a varying sampled input. MSFs can be empowered with additional features by means of different monads, which can serve both to make them more expressive, to synthesize information, or with

different control mechanisms. In the following, we turn to Monadic Stream Functions, and later explain current limitations that this paper addresses.

2.2 Monadic Stream Functions

Monadic Stream Functions (MSFs) are an abstraction for Functional Reactive Programming that supports discrete and continuous time, and both Classic and Arrowized variants.

2.2.1 Fundamental Concepts. MSFs are defined by a polymorphic type *MSF* and an evaluation function that applies an *MSF* to an input and returns, in a monadic context, an output and a continuation:

$$\textbf{newtype}\ \mathit{MSF}\ m\ a\ b$$

$$\mathit{step} :: \mathit{Monad}\ m \Rightarrow \mathit{MSF}\ m\ a\ b \rightarrow a \rightarrow m\ (b, \mathit{MSF}\ m\ a\ b)$$

We purposefully hide the details of the implementation of MSF, which will be shown later on[1]. Functions to build and combine MSFs will be provided in the rest of this section.

The type *MSF* and the *step* function alone do not represent causal functions on streams. It is only when we successively apply the function to a stream of inputs and consume the side effects that we get the unrolled, streamed version of the function. Causality, or the requirement that the n-th element of the output stream only depend on the first n elements of the input stream, is obtained as a consequence of applying the MSF continuations *step by step*, or sample by sample.

For the purposes of exposition, we will use the following function to apply an MSF to a *finite list* of inputs, with effects and continuations chained sequentially. This is merely a debugging aid, not how MSFs are actually executed:

$$\mathit{embed} :: \mathit{Monad}\ m \Rightarrow \mathit{MSF}\ m\ a\ b \rightarrow [a] \rightarrow m\ [b]$$

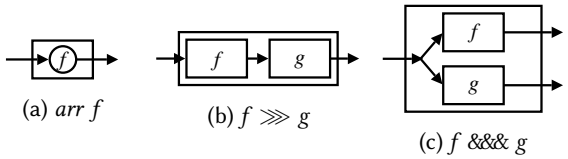
In order to ensure that signal functions are executable, we require them to be *causal*: The output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$.

2.2.2 Composing Monadic Stream Functions. Programming with MSFs consists of defining monadic stream functions compositionally using a library of primitive stream functions and a set of combinators. MSFs are *Arrow*s [Hughes 2000], and so *Arrow* combinators can be used to create them and compose them. Some central *Arrow* combinators are *arr* that lifts an ordinary function to a stateless signal function, composition $\ggg$, parallel composition &&&. In MSFs, they have the following types:

$$\begin{aligned}
&\mathit{arr} &&:: (a \rightarrow b) &&\rightarrow \mathit{MSF}\ m\ a\ b\\
&(\ggg) &&:: \mathit{MSF}\ m\ a\ b \rightarrow \mathit{MSF}\ m\ b\ c &&\rightarrow \mathit{MSF}\ m\ a\ c\\
&(\&\&\&) &&:: \mathit{MSF}\ m\ a\ b \rightarrow \mathit{MSF}\ m\ a\ c &&\rightarrow \mathit{MSF}\ m\ a\ (b, c)
\end{aligned}$$

We can think of streams and monadic stream functions using a simple flow chart analogy. Line segments (or "wires") represent input streams, with arrowheads indicating the direction of flow. Boxes represent MSFs, with one input flowing into the box's input port and another stream function flowing out of the box's output port. Figure 1 illustrates the aforementioned combinators using

[1] The use of the function *step* to define the behaviour of MSFs, as opposed to giving a model implementation, may seem a misleading attempt at appearing rigurous. The purpose of hiding the constructor is simply to limit users to the API that will be provided in the rest of the chapter, which has been designed to preserve causality and avoid leaks. Exposing the data constructor would also make all the following definitions use it explicitly, which would confuse readers and open the door to ill-behaved definitions.



(a) *arr f* (b) $f \ggg g$ (c) *f &&& g*

Fig. 1. Basic MSF combinators.

this analogy. Through the use of these and related combinators, arbitrary MSF networks can be expressed.

2.2.3 Depending on the Past. MSFs functions must remain causal and leak-free, and so MSFs introduce limited ways of depending on past values. To keep state by producing an extra value or *accumulator* accessible in future iterations we use:

$$feedback :: c \to MSF\ m\ (a, c)\ (b, c) \to MSF\ m\ a\ b$$

This combinator takes an initial value for the accumulator, runs the MSF, and feeds the new accumulator back for future iterations.

Example. The following calculates the cumulative sum of its inputs, initializing an accumulator and using a feedback loop:

```
sumFrom :: (Num n, Monad m) ⇒ n → MSF m n n
sumFrom n0 = feedback n0 (arr add2)
  where
    add2 (n, acc) = let n' = n + acc in (n', n')
```

A counter, for example, can be defined as follows:

```
count :: (Num n, Monad m) ⇒ MSF m () n
count = arr (const 1) ⋙ sumFrom 0
```

2.2.4 Monads. MSFs can be combined with different monads for different effects. We provide a general function *arrM* to lift a Kleisli arrow, applied point-wise to every sample:

$$arrM :: Monad\ m \Rightarrow (a \to m\ b) \to MSF\ m\ a\ b$$

The use of monads with MSFs provides great versatility. For example, we can make certain values available in an environment in a *Reader* monad, without having to route them down manually as inputs to other MSFs:

```
data Env = Env { windowWidth  :: Int
               , windowHeight :: Int
               }
rotateMousePos180 :: MSF (Reader Env) (Int, Int) (Int, Int)
rotateMousePos180 = proc (x, y) → do
  winW ← arrM (ask windowWidth)  -≺ ()
  winH ← arrM (ask windowHeight) -≺ ()
  returnA -≺ (winW - x, winH - y)
```

It is possible to "flatten" an MSF by removing the monadic effect, by means of what are called *MSF running functions*. This normally requires extra inputs, extra outputs, or extra continuations. For example, the running function for the reader monad is defined as:

$$runReaderS :: MSF\ (ReaderT\ r\ m)\ a\ b \to r \to MSF\ m\ a\ b$$

ghci > *embed* (*runReaderS* (*rotateMousePos180*) (*Env* 1024 768) [(10, 10), (100, 100)])
[(1014, 758), (924, 668)]

Following the same pattern as before, the associated execution function for terminating *MSF*s would have type:

$$runMaybeS :: Monad\ m \Rightarrow MSF\ (MaybeT\ m)\ a\ b \to MSF\ m\ a\ (Maybe\ b)$$

The evaluation function *step*, instantiated for this particular monad, would have the type $MSF\ Maybe\ a\ b \to a \to Maybe\ (b, MSF\ Maybe\ a\ b)$, indicating that it may produce *no continuation*. The function *runMaybeS* outputs *Nothing* continuously once the internal *MSF* produces no result. "Recovering" from failure requires an additional continuation:

$$catchM :: Monad\ m \Rightarrow MSF\ (MaybeT\ m)\ a\ b \to MSF\ m\ a\ b \to MSF\ m\ a\ b$$

For the case of *Either c*, another form of terminating *MSF*s, this closely resembles the signature of Yampa's *switching* combinators, showing that switching emerges for free in MSF by virtue of combining the *Either* monad. Another possibility is to use a list monad, which gives rise to constructs that maintain *dynamic collections* of monadic stream functions connected in *parallel*.

The first class status of MSFs, in combination with the extensibility provided by different monad transformers and their running functions makes MSFs unusually flexible for describing reactive systems.

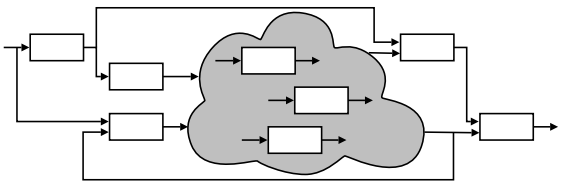


Fig. 2. System of interconnected signal functions with varying structure

2.2.5 Overlap with Streams, Signals, and FRP. The framework described so far is inherently discrete and focused on functions, not streams. However, different elections of inputs, outputs and monads lets us realize streams, monadic streams, Classic FRP signals and Arrowized FRP Signal Functions.

MSFs that do not depend on their input produce a stream of outputs in a monadic context. We can also re-introduce a parameter in a monadic stream by using the reader monad, making MSFs and Monadic Streams equally expressive. When the monad is the identity, monadic streams become plain infinite streams.

Monadic Streams can also be used with monads that include external input and or time in a *Reader* monad, implementing FRP. Using the same idea, we can introduce the explicit time in the environment to implement Arrowized FRP, obtaining a step function that is isomorphic to Yampa's internal representation of initialized Signal Functions. This makes it possible to run Yampa games on top of an intermediate layer that implements an API compatible version of FRP on top of MSFs.

2.3 Faults in Reactive Systems

Let us present an introductory example to illustrate the kinds of issues we seek to address. Imagine that we are building a system to control a spacecraft. To move and orient the spacecraft as required for a mission, we carry out a phase of *position/attitude determination* (determining the position and direction of the spacecraft) prior to *attitude control* (corrections to the orientation via a series of actuators). Errors in these calculations can lead to the spacecraft failing to perform its mission, or permanently damaging its sensors by pointing them in the wrong direction.

To calculate the spacecraft's attitude with the required precision, we combine data from a Star Tracker, which determines the position and orientation relative to stars whose locations in space we know, and from an Inertial Measurement Unit (IMU), which combines gyroscopes and accelerometers to provide an estimate of the spacecraft's acceleration, orientation and surrounding magnetic fields.

A schematic definition of our control system follows. Our reactive control *MSF* takes a desired attitude as input, and produces a set of actions as output. Internally, it gathers data from a star tracker and the IMU. The use of feedback allows us to calculate the new attitude based on the last known attitude and the information gathered from the sensors:

```
controlSystem :: Attitude → MSF m Attitude [Action]
controlSystem initialAttitude = proc (desiredAttitude) → do
  stars       ← starTrackerSense ⪱ ()
  inertialInfo ← imuSense        ⪱ ()
  attitude    ← feedback initialAttitude (arr (dup ∘ calculateAttitude))
                  ⪱ (stars, inertialInfo)
  let actions = calculateActions desiredAttitude attitude
  returnA ⪱ actions
  -- Predefined elsewhere
type Action      = ...
type Attitude    = ...
type StarInfo    = ...
type InertialInfo = ...
starTrackerSense :: MSF m () [StarInfo]
imuSense         :: MSF m () InertialInfo
calculateAttitude :: (([StarInfo], InertialInfo), Attitude) → Attitude
calculateActions  :: Attitude → Attitude → [Action]
dup               :: a → (a, a)
```

In normal conditions, we would work under the assumption that the hardware works perfectly, that values from input devices are accurate, and that the memory is never corrupted. However, this is not true in practice, and critical systems that carry out important operations or function over long periods of time in hostile environments without maintenance, like a satellite, warrant extra levels of reliability.

In particular, in our example, nothing tells us *whether data is inaccurate*, *how inaccurate it can be*, and what *kinds of faults* may have lead to such incorrect results. In cases like these, using a plain *Maybe* type to encode the possibility of failure in the output type of a sensor is not enough.

The use of techniques and extra redundancy to minimize the impact of system failures is grouped under the umbrella term of *fault tolerance* [Avižienis 1967, 1976]. That redundancy can be used

either to compare results from multiple units and disable those that fail, or to average them and lower the impact of minor deviations (a process called *voting*). For instance, in our example, we might require that three star trackers are installed, and that an mid-point between the three is used in calculations, as well as to detect if either star tracker is providing values that are too far from the average.

Apart from adding extra redundancy, and to limit the possibility of multiple redundant units failing for the same reason, we may choose to divide our architecture in independent subsystems, which may be both physically and electrically isolated, constituting independent *Fault Containment Regions (FCRs)*.

The combination of these fault tolerance mechanisms helps determine which of possible faults may still affect system operations, and defines the *fault model*. Total reliability is never possible, so our fault model is likely to determine both the kinds of faults that are handled and also how many simultaneous faults may be tolerated. For instance, a system may be able to deal with a value not being available, but not be with a value being incorrect. Others are capable of handling Byzantine errors, those in which different subsystems have different incorrect values for the same conceptual element, whereas others may only be able to do so, so long as the error is corrected before another simultaneous error occurs.

2.4 Limitations

As we saw before, the use of stream-based frameworks, Monadic Stream Functions and FRP renders simple, clear implementations of reactive system. However, these descriptions operate under the assumption that systems do not fail in unexpected ways and that any erroneous values can be detected, which is not always the case. The limitations that we seek to address are the following[2]:

Availability. System descriptions using MSFs do not help us understand how reliably systems can be and how much time we can expect them to be un/available. While we may have information about the expected availability or reliability of different subsystems or components, gathered from the manufacturer or by prior experience, these implementations hide the details and data dependencies, and do not help synthesize a global reliability factor based on those of expected components.

Ranges and Probabilities. These system descriptions operate with exact values, and, unless randomized testing is used, simulations would normally work with ideal inputs from the sensors. In practice, most components and subsystems have margins of error, and a range of values may be expected. The use of probability distributions constitutes a better mechanism to represent possible input and, potentially, analyse the possibility of values being within the desired margins of tolerance.

Fault Model. These frameworks let us specify data dependencies from multiple subsystems, and therefore hide details in the implementation. In particular, nothing in our example lets us know that attitude calculation may be affected by, for example, condensation in the star tracker's battery unit. The use of monads, available in Monadic Stream Functions, opens an opportunity to specify these failures at the type level, and let them propagate across a network, tagging all reactive constructs that may have been affected.

In the following we present an extension to Monadic Stream Functions that lets us transparently alter the monad to work with parameterized or indexed monads, which will constitute a better mechanism for type-level programming. We show that information can be used in compile time and

[2]In the following, when we refer to MSFs, the limitations apply generally also to other forms of FRP and stream programming.

in runtime, rendering different kinds of analyses. Implementations with slightly different syntax and features are provided both in Haskell and in Idris.

3 RELIABILITY FACTORS

We can try to estimate the reliability that is expected of a reactive MSF system based on the reliability of the subsystems that form it, which may itself be obtained from the manufacturer or from previous experiments.

In order to understand how to work with confidence factors, imagine we gather data from the two sensors we used in the example in Section 2, and want to tag them with a reliability factor. For the sake of simplicity, we use a simple numerical value between 0 and 1 in this section, with 0 expressing *no confidence or reliability* on the accuracy of the value, and 1 expressing *absolute certainty*. In Section 5 we show how to use more structured and realistic representations of reliability.

We can represent this information in MSFs by just changing the inputs and the outputs. First, let us introduce a type for values with a degree of uncertainty:

```
type Uncertain a = (Certainty, a)
type Certainty   = Double   -- zero to one
```

For simplicity, we assume that reliability factors are fixed, irrespective of the conditions of operation. This is unrealistic in practice, but modifying this example to include factors that depend on external conditions would be trivial, and the lack of complexity does not invalidate our claim. We rely on *MSF*s defined earlier and state that both sensors have a 90% accuracy:

```
starTrackerSense' :: MSF m () (Uncertain [StarInfo])
starTrackerSense' = starTrackerSense >>> arr (λstarData → (0.9, starData))

imuSense' :: MSF m () (Uncertain InertialInfo)
imuSense' = imuSense >>> arr (λimuData → (0.9, imuData))
```

These new *MSF*s simply add some extra information to the output. We now adapt both the *step* function and the attitude and action calculation functions to handle uncertainty. Functions like *controlSystemStep'* just route the uncertainty together with the accompanying values:

```
controlSystemStep' :: Uncertain Attitude → MSF m Attitude (Uncertain [Action])
controlSystemStep' initialAttitude = proc (desiredAttitude) → do
  stars       ← starTrackerSense' -≺ ()
  inertialInfo ← imuSense'        -≺ ()
  attitude    ← feedback initialAttitude (arr (dup ∘ calculateAttitude'))
                  -≺ (stars, inertialInfo)
  let actions  = calculateActions' desiredAttitude attitude
  returnA -≺ actions
```

Functions like *calculateAttitude'*, however need access to this factor to calculate the combined uncertainty of the result. If the sensors are independent (e.g., one does not become less accurate when the other fails to report accurate data), then we could say that the result is accurate if both sensors are accurate. Therefore, the combined confidence of a result based on data from two sensors is the multiplication of the reliabilities of both. This operates similar to simple probabilities of independent events.

```
calculateAttitude' :: ((Uncertain [StarInfo], Uncertain InertialInfo), Uncertain Attitude)
                   → Uncertain Attitude
```

```
calculateAttitude' ((starUnc, starData), (inerUnc, inerData)), (attUnc, attData)) =
  let attitude = calculateAttitude ((starData, inerData), attData)
  in  (attUnc * starUnc * inerUnc, attitude)

calculateActions' :: Attitude → Uncertain Attitude → Uncertain [Action]
calculateActions' desiredAttitude (attUnc, attData) =
  (attUnc, (calculateActions desiredAttitude attData))
```

Our main function *controlSystemStep'* now also estimates the confidence on the action. In a hypothetical system, if we execute this action and obtain a value lower than a given threshold, we might want to use an alternative calculation, turn off some devices, or address the loss of accuracy in some other way.

Of course, passing these factors by hand is error prone. Without a facility to guarantee that calculations are correct, not only does this method not increase the knowledge we have about certainty in our system, a false sense of security placed on potentially incorrect results.

Luckily, we can abstract the underlying details into a monad that performs the calculation steps taking the certainty factors into account. This monad is exactly like a *Writer* monad, in which the monoid are real numbers between 0 and 1 in the log, using multiplication as the monoidal operation and 1 as identity. However, monoidal operations would only maintain or decrease certainty (i.e., multiplication of a positive value by a value between 0 and 1 can render it smaller, but not larger). In our case, since we want to provide fault-tolerance, we also need a way to *increase* confidence on our system, which requires that we inspect the contents of the *Writer*.

The following section captures this pattern of operation with a custom monad, which we later enrich with further structure to introduce fault-tolerance mechanisms.

3.1 Fault Tolerance Monad

Let us now abstract the details of the previous example into a type for "uncertain" or "imprecise" values. As we saw before, this is just a polymorphic product type, in which we tag values with extra reliability information:

```
data FaultTolerance e a = FaultTolerance
  { toleranceInfo  :: e
  , toleranceValue :: a
  }
```

In order to operate with these values and hide the operational details from the user, let us define a suitable monad. To work with these factors, we will impose additional constraints on some operations.

For monadic sequencing of operations, we first need to be able to combine faults, so they must have a binary operation with identity, making them a *monoid*. While this structure is equivalent to a *Writer* monad, in practice we also require additional constraints on *e* (namely, that it be a non-commutative *ring*) to implement general mechanisms for fault tolerance. This is discussed in Section 11.

Obviously, *FaultTolerance* is a *Functor*, as we can always apply a transformation that does affect the (un)reliability of a value:

```
instance Functor (FaultTolerance f) where
  fmap f (FaultTolerance factor value) = FaultTolerance factor (f value)
```

The rules needed to operate with these factors are captured by the *Applicative* and *Monad* instances, which require a *Monoid* type constraint. The implementation is similar for both: lifting pure values assumes a neutral uncertainty (*mempty*), and combining certainties uses the binary *Monoid* operation (*mappend*):

```
instance Monoid f ⇒ Applicative (FaultTolerance f) where
  pure x      = FaultTolerance mempty x
  f <*> x = FaultTolerance (toleranceInfo f `mappend` toleranceInfo x)
                           (toleranceValue f $ toleranceValue x)
instance Monoid f ⇒ Monad (FaultTolerance f) where
  return  = pure
  m >>= f = FaultTolerance (factor1 `mappend` factor2) value2
    where
      FaultTolerance factor1 value1 = m
      FaultTolerance factor2 value2 = f value1
```

To hide details from the user, we add an operation to put a value with a specific tag:

```
liftUnreliable :: e → a → FaultTolerance e a
liftUnreliable e x = FaultTolerance e x
```

3.2 Example

Adapting the previous example to this new interface should be straightforward, and it only makes the code simpler (we overload names where appropriate):

```
starTrackerSense' :: MSF (FaultTolerance Certainty) () [StarInfo]
starTrackerSense' = starTrackerSense >>> arrM (liftUnreliable 0.9)

imuSense' :: MSF (FaultTolerance Certainty) () InertialInfo
imuSense' = imuSense >>> arrM (liftUnreliable 0.9)
```

We adapt the main control functions as follows. For the sake of simplicity, we temporarily eliminate the monad *m* from the stack. We remedy this situation later with the introduction of a monad transformer for fault tolerance.

```
controlSystemStep' ::  Attitude
                    → MSF (FaultTolerance Certainty) Attitude [Action]
controlSystemStep' initialAttitude = proc (desiredAttitude) → do
  stars       ← starTrackerSense' -< ()
  inertialInfo ← imuSense'        -< ()
  attitude    ← feedback initialAttitude (arr (dup ∘ calculateAttitude))
                   -< (stars, inertialInfo)
  actions     ← arr (uncurry calculateActions) -< (desiredAttitude, attitude)
  returnA -< actions
```

Note that this only changes the signature of the function, but not the implementation. This is a key strength of this approach, and we explore this benefit further in future sections.

Calculating the attitude and the actions now becomes trivially simple and we can use the original functions, as it is leveraged onto the *Monad* instance of *FaultTolerance*.

We can generalise this example to make it extensible and even simpler by means of monad transformers [Liang et al. 1995a], as we present next.

3.3 Fault Tolerance Monad Transformer

We can generalize the previous type into a transformer that allows us to combine it with other computations. The type definition simply encapsulates the fault tolerant value in a monad:

```
data FaultToleranceT e m a = FaultToleranceT
  { runFaultToleranceT :: m (e, a) }
```

As usual with transformers, we have that **type** *FaultTolerance* = *FaultToleranceT Identity*. The implementation is almost the same as before, except that, in this case, the type constraints are imposed on the internal monad and the composed monad, which makes the type signatures slightly more complicated. We take advantage of the *Functor*, *Applicative* and *Monad* instances of pairs with a fixed first component to simplify the implementation.

The functor instance is simply the function applied to the second element of type *a* in the pair:

```
instance Functor f ⇒ Functor (FaultToleranceT e f) where
  fmap f (FaultToleranceT m) = FaultToleranceT (fmap (λ(e, v) → (e, f v)) m)
```

The combination of effects uses the *Monoid* instance of the confidence factor:

```
instance (Monoid e, Applicative f, Functor (FaultToleranceT e f))
         ⇒ Applicative (FaultToleranceT e f) where
  pure x = FaultToleranceT < $ > pure (mempty, x)
  (FaultToleranceT f) < ∗ > (FaultToleranceT x) = d
    FaultToleranceT $
      ((ff, fv) (xf, xv) → (ff `mappend` xf, fv xv)) < $ > f < $ > x
```

Similarly, the combination of effects in the *Monad* instance uses the *Monoid* instance to combine the factors:

```
instance (Monoid e, Monad m, Applicative (FaultToleranceT e m))
         ⇒ Monad (FaultToleranceT e m) where
  return = pure
  (FaultToleranceT m) >>= (FaultToleranceT f) =
    FaultToleranceT $ do (xf, xv) ← m
                         (yf, yv) ← f xv
                         return (xf `mappend` yf, yv)
```

We provide two convenience functions to create values with limited and with absolute confidence:

```
liftUnreliableT :: Monoid e ⇒ m a → FaultToleranceT e m a
liftUnreliableT f m = FaultToleranceT < $ > (λx → (f, x)) < $ > m
liftReliableT :: Monoid e ⇒ m a → FaultToleranceT e m a
liftReliableT m = FaultToleranceT < $ > (λx → (mempty, x)) < $ > m
```

3.4 Example

Our main program remains largely the same, except that we must use the new operations in our calculations:

```
starTrackerSense' :: MSF (FaultToleranceT Certainty m) () [StarInfo]
starTrackerSense' = starTrackerSense >>> arrM (liftUnreliableT 0.9)

imuSense' :: MSF (FaultToleranceT Certainty m) InertialInfo
imuSense' = imuSense >>> arrM (liftUnreliableT 0.9)
```

We now simplify the main control function:

```
controlSystemStep' ::  Attitude
                    -> MSF (FaultToleranceT Certainty m) Attitude [Action]
controlSystemStep' initialAttitude = proc (desiredAttitude) -> do
  stars        <- starTrackerSense' -< ()
  inertialInfo <- imuSense'         -< ()
  attitude     <- feedback initialAttitude (arr (dup . calculateAttitude))
                    -< (stars, inertialInfo)
  let actions  =  calculateActions desiredAttitude attitude
  returnA -< actions
```

The auxiliary functions *calculateAttitude′* and *calculateActions′* are now unnecessary, as their fault tolerance handling is now embedded in the monad. This last example demonstrates the elegance of this solution, we have incorporated uncertainty in our function without making it any more complex than it originally was.

In the following section we will see how this can be expanded on, by adding more structured information in our uncertainty factors.

4 FAULT TOLERANCE AND VOTING

Fault Tolerance is usually implemented by introducing *redundancy*, that is, multiple units of the same kind that perform the same actions. This redundancy can be used either to detect failures (by comparing the results of different redundant units), to mask failures (by averaging the results of different redundant units), or to recover from failure (by discarding faulty units and replacing them by their redundant counterparts).

In the previous section, we saw how to calculate using values from multiple, potentially unreliable sensors. Because certainty is always between 0 and 1 and we use the product as the monoidal operation, it can remain stable and decrease, but never increase.

We can address this limitation by adding new functions that manipulate data from different sensors, and produce a result with higher certainty.

As an example, let us consider the possibility (not necessarily realistically) that a satellite has three star trackers, and tries to use the extra information to calculate an average position. We obtain data from all sensors, together with tolerance factors, and determine that this approach is reliable so long as only one star tracker fails. We use $\lambda x \rightarrow 1 - x$ to invert reliability factors, and calculate as follows:

```
starTrackerAvg' :: MSF (FaultToleranceT Certainty m) () StarInfo
                -> MSF (FaultToleranceT Certainty m) () StarInfo
                -> MSF (FaultToleranceT Certainty m) () StarInfo
```

```
                  → MSF (FaultToleranceT Certainty m) () StarInfo
starTrackerAvg' starInfo1 starInfo2 starInfo3
  = proc () → do
      (f1, d1) ← runFaultToleranceS starInfo1 -≺ ()
      (f2, d2) ← runFaultToleranceS starInfo2 -≺ ()
      (f3, d3) ← runFaultToleranceS starInfo3 -≺ ()

        -- A receiver fails if it does not work
      let fails1  = 1 - f1
          fails2  = 1 - f2
          fails3  = 1 - f3

        -- Two fail if any combination of two receivers fail
      let twoFail = fails1 * fails2 + fails2 * fails3 + fails1 * fails3

        -- At least two work if two don't fail
      let twoWork = 1 - twoFail

      returnA -≺ arrM (λ() → liftUnreliableT twoWork (starTrackerAvg d1 d2 d3))
starTrackerAvg :: StarInfo → StarInfo → StarInfo → StarInfo
starTrackerAvg = ...
```

Note that, in this case, we enclose the Star Tracker polling operations by the function *starTrackerAvg'*, using a plan MSF that gets the data from each sensor as stream input and averages it. This is because we want the reliability factors of the Star Tracker data not to influence our result directly, and denotes a *fault tolerance region* (FCR).

Our fault tolerant control step MSF is now slightly different, as it needs to pass three star trackers as input to this function:

```
controlSystemStep' ::  Attitude
                    → MSF (FaultToleranceT Certainty m) Attitude [Action]
controlSystemStep' initialAttitude = proc (desiredAttitude) → do
  stars        ← starTrackerAvg' starTrackerInfo1'
                                 starTrackerInfo2'
                                 starTrackerInfo3' -≺ ()
  inertialInfo ← imuSense' -≺ ()
  attitude     ← feedback initialAttitude (arr (dup ∘ calculateAttitude))
                    -≺ (stars, inertialInfo)
  let actions  =  calculateActions desiredAttitude attitude
  returnA -≺ actions
```

Running this new experiment shows that our confidence in the result has increased due to the more reliable fault tolerance mechanism:

```
ghci > embed (runFaultTolerantS (controlSystemStep initialAttitude))
             [destAttitude, destAttitude, destAttitude]
[(0.9603, [...]), (0.9603, [...]), (0.9603, [...])]
```

This approach could be generalized to perform other kinds of recovery mechanisms. For example, we could make *starTrackerAvg* operate over lists, and ignore trackers whose current values are too far from the average, increasing the overall reliability factors.

5 PROBABILITY

Let us now enrich the type we are using to represent confidence with further structure, indicating a range of possible values and their probabilities.

We resort to an existing representation for probabilities [Erwig and Kollmansberger 2006], which we briefly introduce below. We later explore how to combine this system with Monadic Stream Functions to obtain reactive specification with probability distributions.

5.1 Probabilities in Haskell

The representation of a type for probabilities has been the subject of prior study in Haskell, and it is not our goal to define a custom type for probabilities but, rather, to show that with few properties we can combine existing, orthogonal representations of probability distributions with reactive programming. In this paper, we follow the description in [Erwig and Kollmansberger 2006] due to conciseness and elegance. We only require that we can construct a monad, and leave aside the discussion of which specific representation to use.

We first introduce an abstract type constructor T that represents a probability distribution. Details of T's internal definition is purposefully hidden:

$$\textbf{newtype}\ T\ prob\ a$$

For example, the type $T\ Double\ Bool$ represents the distribution of an event being *True* or *False*, expressed as probabilities with type *Double*. This flexibility lets us use alternative types with different precision to represent probabilities.

For clarity, we define:

$$\begin{aligned}\textbf{type}\ Dist &= T\ Probability\\ \textbf{type}\ Probability &= Rational\end{aligned}$$

We can represent values with different probability distributions using multiple auxiliary functions. For example, we can represent a uniform distribution as follows:

$$\begin{aligned}&die :: Dist\ Int\\ &die = uniform\ [1\,..\,6]\end{aligned}$$

This value can be examined directly in a session[3], showing the spread of the distribution with the probability of each individual event:

$$\begin{aligned}&ghci > die\\ &fromFreqs\ [(1, 0.16), (2, 0.16), (3, 0.16), (4, 0.16), (5, 0.16), (6, 0.16)]\end{aligned}$$

We can also query the probability of individual events, for instance, to obtain the probability of obtaining more than 4 if we throw a die:

$$\begin{aligned}&ghci > (>4)\ ??\ die\\ &0.33\end{aligned}$$

[3]The numbers in this GHCi session are cropped to two decimals for readability.

We can also create normal distributions, or distributions from specific lists of events and frequencies:

```
height :: Dist Double
height = normal [55, 60, 65, 70, 75, 80, 85]

probSuccess :: Dist Bool
probSuccess = fromFreqs [(True, 0.55), (False, 0.45)]
```

Many other auxiliary functions, types and distributions are available in this library. A crucial observation in the following is that *T prob* is a monad if *prob* is a number, which means that it can be used in place of *FaultTolerant* to capture different possible results of reactive components, and their probabilities, as we show in the following.

5.2 Probabilities in Reactive Systems

Since *T prob* is a *Monad* if *prob* is a number, we can immediately use it together with MSFs. We can use probabilities with MSFs is to actually modify or alter the value, introducing noise in our signals. For example, we can introduce values slightly different from that of an ideal star tracker (assuming we can multiply it). We indicate that the probability of those differences is 0.1%, while we have a probability of 99.6% of obtaining the exact value:

```
starTrackerSense' :: MSF Dist () [StarInfo]
starTrackerSense' = starTrackerSense >>> arrM
  (λs -> let [s1, s2, s3, s4] = map (adjustStarInfo s) [0.90, 0.95, 1.05, 1.10]
    in fromFreqs [(s1, 0.001), (s2, 0.001), (s3, 0.001), (s4, 0.001), (s, 0.996)])
  -- Defined elsewhere
adjustStarInfo :: Num a => [StarInfo] -> a -> [StarInfo]
```

Just like before, adapting all operations makes our types change, but our main function remains unchanged:

```
controlSystemStep' :: Attitude -> MSF Dist Attitude [Action]
controlSystemStep' initialAttitude = proc (desiredAttitude) -> do
  stars       <- starTrackerSense' -< ()
  inertialInfo <- imuSense'        -< ()
  attitude    <- feedback initialAttitude (arr (dup . calculateAttitude))
                   -< (stars, inertialInfo)
  let actions  = calculateActions desiredAttitude attitude
  returnA -< actions
```

To explore values inside *Dist*, we define an auxiliary function, *runDistS*, that extracts the values of an MSF in the *Dist* monad, putting the probabilities in the output[4].

If we run this in a session, we get a probability distribution of all possible outputs (formatted for readability):

```
ghci > embed (runDistS $ controlSystemStep' initialAttitude) [destAttitude]
[([Action...], 0.001), ([Action...], 0.001), ([Action...], 0.001)
, ([Action...], 0.001), ([Action..., 0.996])
```

[4]Internally, this particular monad is defined similarly to the so-called "broken" list transformer *ListT*. Since MSFs already support working with *ListT*, defining this function is trivial and we obviate that detail.

```
]
```

An second possibility is to use probabilities to tag the possibility of errors taking place. This will be discussed later in Section 6.

5.3 Fault Tolerance with Probabilities

Due to the way that probabilities are represented here, adding fault tolerance in this case is easier than with other constructs. The definitions of *bind* and *return* for the *Dist* monad already calculate the new probabilities of the different possible combinations of inputs and outputs, which makes it easier to write expressions that combine probabilities. In this case, manually calculating the inverses is not necessary, and an occurrence of the same event multiple times will be combined to provide an accumulated event with the addition of both probabilities.

```
starTrackerAvg' :: MSF Dist () StarInfo
                → MSF Dist () StarInfo
                → MSF Dist () StarInfo
                → MSF Dist () StarInfo
starTrackerAvg' starInfo1 starInfo2 starInfo3
  = proc () → do
      d1 ← starInfo1 -≺ ()
      d2 ← starInfo2 -≺ ()
      d3 ← starInfo3 -≺ ()
      returnA -≺ starTrackerAvg d1 d2 d3
starTrackerAvg :: StarInfo → StarInfo → StarInfo → StarInfo
starTrackerAvg = ...
```

If we now explore the probabilities with averaging, we see that it is more likely to be wrong but (it will only be perfectly correct if all star trackers work perfectly) but, due to the average eliminating part of the error, it would be normally wrong by a lower amount.

```
ghci > embed (runDistS $ controlSystemStep' initialAttitude) [destAttitude]
[(Action ..., 0.988047936), (Action ..., 2.985015e−3), (Action ..., 2.982025e−3)
, (Action ..., 2.979036e−3), (Action ..., 2.976048e−3), (Action ..., 1.1958e−5)
...
]
```

6 FAULT SETS AND FAULT ANALYSIS

While the previous approach is informative to get a general understanding of the availability or reliability we could expect from our system, it does not help us determine *what caused the failure*.

Let us now consider a more interesting case in which we will use the same abstraction to indicate, in the types, the possible faults that may have affected the calculations. This is very similar to what one captures normally with an *Either* type, except that *Either* assumes that we known when a value is incorrect. In fault tolerance there is a class of faults that we cannot or have not detected, termed *transmissive*, which are passed to upper layers. For example, a bit flip in the memory of the star tracker may corrupt the results, and, if we are building, for example, a data transmission system for the satellite, we may not have sufficient application knowledge to understand which values may be completely wrong.

Even if we do not handle a particular kind of fault at a level, we might want to have a clear, machine-checked specification of faults that we handle and those we do not. Let us first define a type that contemplates all possible faults in our *fault model*:

```
data Fault = StarTrackerBatteryError
           | StarTrackerFixNotFound
           | StarTrackerProcessingError
           | ...
  deriving (Ord, Eq)
```

We now use sets with union and the empty set as the monoidal operations for the Fault Tolerance monad, so our *starTrackerSense′* function becomes:

```
starTrackerSense' :: MSF (FaultToleranceT (Set Fault) IO) () StarInfo
starTrackerSense' = starTrackerSense >>> arrM
                      (liftUnreliableT
                        [StarTrackerBatteryError
                        ,StarTrackerFixNotFound
                        ,StarTrackerProcessingError
                        ]
                      )
```

We assume that *imuSense′* changes accordingly. Like in previous occasions, our main function changes its type, but the implementation remains the same. This demonstrates a key strength of MSFs in their use to specify and extend reactive systems with additional features:

```
controlSystemStep' :: Attitude → MSF (FaultToleranceT (Set Fault) m) Attitude [Action]
controlSystemStep' initialAttitude = proc (desiredAttitude) → do
  stars       ← starTrackerSense' -< ()
  inertialInfo ← imuSense'        -< ()
  attitude    ← feedback initialAttitude (arr (dup ∘ calculateAttitude))
                  -< (stars, inertialInfo)
  let actions  = calculateActions desiredAttitude attitude
  returnA -< actions
```

We now run one step of this function to obtain a list of all possible faults that may have affected the result. Due to the way that the monad affects all MSFs connected to the *starTrackerSense′*, the top level MSF also includes all the faults that faults that may have affected the star tracker:

```
ghci > embed (runFaultToleranceS $ controlSystemStep' initialAttitude) [destAttitude]
[(Set [StarTrackerBatteryError, StarTrackerFixNotFound, StarTrackerProcessingError],
  ,(Action ...))]
```

In Section 4 we introduced voting to help eliminate some classes of faults. Let us adapt that example to this new case, to show the benefits of adding explicit faults:

```
starTrackerAvg' :: MSF (FaultToleranceT (Set Fault) m) () StarInfo
                → MSF (FaultToleranceT (Set Fault) m) () StarInfo
                → MSF (FaultToleranceT (Set Fault) m) () StarInfo
                → MSF (FaultToleranceT (Set Fault) m) () StarInfo
```

```
starTrackerAvg' starInfo1 starInfo2 starInfo3
  = proc () → do
      (f1, d1) ← runFaultToleranceS starInfo1 -≺ ()
      (f2, d2) ← runFaultToleranceS starInfo2 -≺ ()
      (f3, d3) ← runFaultToleranceS starInfo3 -≺ ()
        -- A receiver fails if it does not work
      let handled = [StarTrackerBatteryError
                    , StarTrackerFixNotFound
                    , StarTrackerProcessingError
                    ]
      let faults1 = f1 \ handled
          faults2 = f2 \ handled
          faults3 = f3 \ handled
      let faults = faults1 `union` faults2 `union` faults3
      returnA -≺ arrM (λ() → liftUnreliableT faults (starTrackerAvg d1 d2 d3))
starTrackerAvg :: StarInfo → StarInfo → StarInfo → StarInfo
starTrackerAvg = ...
```

Running an adapted version of *controlSystemStep′* with this implementation would no longer include possible star tracker faults in the output. However, to make sure we do not remove faults that we do not handle by means of the average, we explicitly list the faults that this method may deal with and put them in the output.

Because *FaultToleranceT* is a monad transformer, we can combine this method with previous ones. For instance, we could state that a value may fail, or, separately, that it may be slightly inaccurate, by using the monad *FaultToleranceT* (*Set Fault*) *Dist*. We could also use a multiset or a more complex structure to include multiple possible errors of the same kind, and state that the fault tolerance mechanisms can help recover from up to a number of possible simultaneous faults, but not more. We have decided to include a fixed fault set, as opposed to injecting faults dynamically in the result. This helps us examine a worst-case scenario, which is what we need for the kind of analysis we are interested in. Of course, this might change if some faults are not possible at particular times or under certain conditions.

7 PARAMETERIZED MONADIC STREAM FUNCTIONS

The approach presented in the previous section allows us to conclude an execution with a set of faults. While this is useful, it does not provide any static guarantees about the presence or absence of certain possible faults. Such information is only obtained during execution.

An alternative approach would be encode specific faults at the type level. We can take advantage of dependently typed programming and encode the possible faults at a particular point using a *Set* of *Faults*. In Haskell, existing packages like type-level sets[5] make this possible [Orchard and Petricek 2015].

For example, in our case, we might have wanted to type *starTrackerSense′* as follows:

```
starTrackerSense' :: MSF
                       (FaultToleranceT IO
                         (Set' [StarTrackerBatteryError, StarTrackerFixNotFound
```

[5] https://hackage.haskell.org/package/type-level-sets

$$, \mathit{StarTrackerProcessingError}\,]))$$
$$()\ \mathit{StarInfo}$$

While this is possible by means of the aforementioned libraries and with multiple type-dependent extensions in GHC, a crucial observation is that the monad changes. In order to make this work for MSFs, we need to extend them to work for indexed or parametric monads. In the rest of this section, we extend MSFs accordingly, and we return to this point in the next section to show the implementation of this function with type-level fault information.

Let us introduce a new interface to work with Monadic Stream Functions that extend the functions presented before. For the purposes of understanding, we also introduce a **newtype** with the definition of MSF, which we use to give model implementations of the new operations available. In the rest of the discourse, we assume that this new interface substitutes the previous one.

7.1 Parametric Monads

If we specify the set of possible faults for an operation using a closed set, then the *Monad* will change if we specify a different set of possible faults. To work around this limitation[6], we opt for *parameterized monads*, which relax the definition of bind so that the monad can change. The definition of the type class for parameterized monads is divided in two type classes:

$$\begin{aligned}&\textbf{class}\ \mathit{Return}\ m\ \textbf{where}\\&\quad \mathit{returnM} :: a \to m\ a\\&\textbf{class}\ \mathit{Bind}\ m1\ m2\ m3 \mid m1\ m2 \to m3\ \textbf{where}\\&\quad \mathit{bindM} :: m1\ a \to (a \to m2\ b) \to m3\ b\end{aligned}$$

Multiple authors have explored different monadic notions with different levels of parametrization, different kinds of constraints, and different categorical meaning. Our goal is to show that, if the monad is allowed to change, we can easily work with type-level sets of faults, which addresses many of our concerns. In practice, some of these proposals are too general, and type systems have trouble giving type signatures to intermediate constructs. We have experimented with different approaches to encode this level of flexibility and we have had most success with *effect monads*, which add an additional parameter to the monad *m*, instead of allowing bind to alter the monad fully. For the rest of the chapter, we describe our proposal based on these parameterized monads, also known as *polymonads*. Other proposals are discussed in Section 9.

7.2 Parametric MSFs

7.2.1 Basic Definitions. Like before, an *MSF* is a type that represents a step in a synchronous causal transformation between changing inputs:

$$\textbf{newtype}\ \mathit{MSF}\ m\ a\ b = \mathit{MSF}\ \{\mathit{step} :: a \to m\ (b, \mathit{MSF}\ m\ a\ b)\}$$

The function *step* takes an input sample and returns, in a monadic context, an output and a continuation. In general, we assume that *m* is an parameterized *Monad*. While the above type exposes the constructor of *MSF*s, in general we discourage users from using it and provide the following leak-free causal interface to define and combine *MSF*s.

In this section, we re-define MSFs using the same names that we used before for functions and combinators. This lets us use the same notation, including arrow **proc** notation. In practice, support

[6]We could have opted to specify type-level fault sets as constraints (Set *s* contains fault *Y*, as opposed to set is $Set'[\,Y\,]$. While this would make the use of parametric monads unnecessary, it would also complicate type signatures.

for re-binding arrow combinators in GHC is limited. Our real implementation simply uses different names. This simplification does not alter the validity of the underlying of our claims.

7.2.2 Point-wise transformations. *MSF*s can be transformed by applying a point-wise transformation on the input. While normally this would be Kleisli arrow for a given *Monad* instance, we use the type classes *Return* and *Bind* to define this function so that the MSF works for parameterized monads:

$$
\begin{aligned}
&\mathit{arrM} :: (\mathit{Return}\ m, \mathit{Bind}\ m\ m\ m) \Rightarrow (a \to m\ b) \to \mathit{MSF}\ m\ a\ b \\
&\mathit{arrM}\ f = \mathit{MSF}\ \$\ \lambda a \to \mathbf{do} \\
&\quad b \leftarrow f\ a \\
&\quad \mathit{returnM}\ (b, \mathit{arrM}\ f)
\end{aligned}
$$

Obviously, one can always apply a pure transformation on the input, so we define, for convenience:

$$
\begin{aligned}
&\mathit{arr} :: (\mathit{Return}\ m, \mathit{Bind}\ m\ m\ m) \Rightarrow (a \to b) \to \mathit{MSF}\ m\ a\ b \\
&\mathit{arr}\ f = \mathit{arrM}\ (\mathit{return} \circ f)
\end{aligned}
$$

7.2.3 Composition. We also provide a way to compose *MSF*s. This is one of the main differences with respect to the previous framework, since the monad now changes:

$$
\begin{aligned}
&(\ggg) :: (\mathit{Return}\ m1, \mathit{Return}\ m2, \mathit{Return}\ m3, \mathit{Bind}\ m1\ m2\ m3, \mathit{Bind}\ m2\ m2\ m2) \\
&\qquad \Rightarrow \mathit{MSF}\ m1\ a\ b \\
&\qquad \to \mathit{MSF}\ m2\ b\ c \\
&\qquad \to \mathit{MSF}\ m3\ a\ c \\
&(\ggg)\ (\mathit{MSF}\ \mathit{msf1})\ (\mathit{MSF}\ \mathit{msf2}) = \mathit{MSF}\ \$\ \lambda a \to \mathbf{do} \\
&\quad (r1, \mathit{msf1}') \leftarrow \mathit{msf1}\ a \\
&\quad (r2, \mathit{msf2}') \leftarrow \mathit{msf2}\ r1 \\
&\quad \mathit{returnM}\ (r2, \mathit{msf1}' \ggg \mathit{msf2}')
\end{aligned}
$$

7.2.4 Widening. We also define ways of applying a transformation to only one input in a pair, leaving the other input unchanged. This definition is straightforward, and follows the one presented earlier, except that it is specialized for the interface of parameterized monads:

$$
\begin{aligned}
&\mathit{first}\ :: (\mathit{Return}\ m, \mathit{Bind}\ m\ m\ m) \Rightarrow \mathit{MSF}\ m\ a\ b \to \mathit{MSF}\ m\ (a, c)\ (b, c) \\
&\mathit{first}\ (\mathit{MSF}\ \mathit{msf1}) = \mathit{MSF}\ \$\ \lambda(a, c) \to \mathbf{do} \\
&\quad (b, \mathit{msf1}') \leftarrow \mathit{msf1}\ a \\
&\quad \mathit{returnM}\ ((b, c), \mathit{first}\ \mathit{msf1}')
\end{aligned}
$$

We can define *second* analogously, although it is not primitive and can also be defined in terms of *first* as $\mathit{second}\ \mathit{msf} = \mathit{arr}\ \mathit{swap} \ggg \mathit{firstM}\ \mathit{msf} \ggg \mathit{arr}\ \mathit{swap}$.

7.2.5 Depending on the Past. Finally, and just like before, we provide a way of creating a well-formed feedback loop:

$$
\begin{aligned}
&\mathit{feedback} :: (\mathit{Return}\ m, \mathit{Bind}\ m\ m\ m) \Rightarrow c \to \mathit{MSF}\ m\ (a, c)\ (b, c) \to \mathit{MSF}\ m\ a\ b \\
&\mathit{feedback}\ c\ (\mathit{MSF}\ \mathit{msf}) = \mathit{MSF}\ \$\ \lambda a \to \mathbf{do} \\
&\quad ((b', c'), \mathit{msf}') \leftarrow \mathit{msf}\ (a, c) \\
&\quad \mathit{return}\ (b', \mathit{feedback}\ c'\ \mathit{msf}')
\end{aligned}
$$

We can also delay the input by one sample, which can be trivially defined in terms of well-formed feedback:

```
iPre :: (Return m, Bind m m m) ⇒ a → MSF m a a
iPre a0 = feedback a0 (arr swap)
  where
    swap (x, y) = (y, x)
```

If we make $m1 \equiv m2 \equiv m3$ in the definitions presented in this section, then the *Return* and *Bind* type constraints simply correspond to standard *Monad* type-class constraints and we obtain an interface equivalent to the one provided before for ordinary Monadic Stream Functions.

8 ENCODING FAULTS AT THE TYPE LEVEL

Now that we have extended MSFs to work with different monads, let us come back to this point and present how to extend the given framework to operate with type-level fault sets.

Throughout the rest of the chapter, we simplify some functions, eliminating trivial constraints from the type signatures. This is limited only to equivalences that can be derived from the monoid laws.

First, we define monad instances for *FaultToleranceT* on fault sets, using the definition of sets as a monoid with union and the empty set as the identity:

```
instance Monad m ⇒ Return (FaultToleranceT (Set'[ ]) m) where
  returnM x = return (Empty, x)
instance Monad m ⇒ Bind (FaultToleranceT (Set x) m)
                        (FaultToleranceT (Set y) m)
                        (FaultToleranceT (Set (x `union` y)) m) where
  bindM (set1, v1) f = do (set2, v2) ← f v1
                          return (set1 `union` set2, v2)
```

Example. We use the same definition of *Fault* as before. Making *Fault*s into type-level *Fault*s requires the use of several dependent-type programming extensions in GHC that promote values to the type level. The use of *Fault* with set requires the definition of an instance of *Cmp*, a type-level comparison function akin to *compare* for instances of *Ord*, used to eliminate duplicates from the set. For convenience, we also define instances for *Show*, to inspect specifications using a GHCi session.

In order to represent these instances at the type level in sets, we need to create a *proxy* type. This lets us create values of those faults to fill in the type-level set. While we could opt to use a generic **data** *Proxy a* = *Proxy* definition, the use of a GADT with multiple value constructors helps the type system deduce the types of type-level sets, making our specifications potentially more robust.

```
data F (a :: Fault) where
  MkStarTrackerBatteryError     :: F StarTrackerBatteryError
  MkStarTrackerFixNotFound      :: F StarTrackerFixNotFound
  MkStarTrackerProcessingError :: F StarTrackerProcessingError
```

We can now express that an operation is supposed to produce a type-level fault. For convenience, we define a type that makes our signatures slightly shorter:

```
type StarTrackerFaultModel = Set'[ F StarTrackerBatteryError
                                 , F StarTrackerFixNotFound
                                 , F StarTrackerProcessingError
                                 ]
starTrackerSense' :: MSF (FaultToleranceT StarTrackerFaultModel m)
                         () [StarInfo]
starTrackerSense' = starTrackerSense >>> arrM
                         (liftUnreliableT (asSet'[MkStarTrackerBatteryError,
                                                  , MkStarTrackerFixNotFound,
                                                  , MkStarTrackerProcessingError
                                                  ]))
```

We can now adapt our main control function accordingly. Once again, this shows

```
controlSystemStep' :: Attitude
                   -> MSF (FaultToleranceT StarTrackerFaultModel m) () [Action]
controlSystemStep' initialAttitude = proc (desiredAttitude) -> do
  stars       <- starTrackerSense' -< ()
  inertialInfo <- imuSense'        -< ()
  attitude    <- feedback initialAttitude (arr (dup . calculateAttitude))
                   -< (stars, inertialInfo)
  let actions  = calculateActions desiredAttitude attitude
  returnA -< actions
```

Type-level Fault Tolerance. Recovering from fault sets now requires eliminating elements from the set. Due to our definition of *Monad* being based on the union of elements, we need to pass the operations as arguments to another function in the same way we did before. This operation is very similar to the one we defined for value-level sets before, except that it uses the API available for type-level sets:

```
starTrackerAvg' :: FaultToleranceT StarTrackerFaultModel m StarInfo
                -> FaultToleranceT StarTrackerFaultModel m StarInfo
                -> FaultToleranceT StarTrackerFaultModel m StarInfo
                -> FaultToleranceT (Set'[]) m StarInfo
starTrackerAvg' starInfo1 starInfo2 starInfo3
  = FaultToleranceT $ do
      (f1, d1) <- runFaultToleranceT starInfo1
      (f2, d2) <- runFaultToleranceT starInfo2
      (f3, d3) <- runFaultToleranceT starInfo3
        -- Should be empty
      let fs = (f1 `union` f2 `union` f3) `delete`
                 '[MkStarTrackerBatteryError
                  , MkStarTrackerFixNotFound
                  , MkStarTrackerProcessingError
                  ]
      return (fs, starTrackerAvg d1 d2 d3)
```

$$starTrackerAvg :: StarInfo \rightarrow StarInfo \rightarrow StarInfo \rightarrow StarInfo$$
$$starTrackerAvg = \ldots$$

Note that, in this case, we assume that faults of the same kind but different origins (i.e., from different redundant devices) be represented differently. If this is not the case, then the type should be modified to allow for similar faults from different origins, either with arguments in the type constructor or by using a type-level multiset.

The types of MSFs reflect the precise sets of faults that need to be considered. In particular, the compiler would throw an error if we indicate in the type of an MSF that it needs not consider a fault if an internal MSF does need it. This is a key strength of this framework, as it lets us know, precisely, what can fail, and why. Together with the system of probabilities introduced before, this allows us to specify not only tolerance ranges of our operations and their probabilities of occurrence, but also the reasons for certain values to be out of their normal ranges and to require specific operations to recover from those failures.

9 IMPLEMENTATION

We have implemented the extension of Monadic Stream Functions both in Haskell and in Idris. The Haskell implementation is, in practice, more useful, due to the possibility of combining it with existing backends, testing frameworks, and other systems. However, part of our goal is to prove system correctness and adherence to specifications and requirements, and we have implemented this framework also in Idris to explore how much information can be added at the type level and what system properties can be proved correct by construction.

Haskell. The implementation in Haskell builds on a larger body of previously existing work, which makes it potentially more useful in practice. Due to the fact that these parameterized *MSF*s subsume the already existing abstraction for MSFs, it makes it possible to define the latter as a special case of the former, making it also possible to test our work with previously existing programs. Furthermore, because Yampa can also be built on top of MSFs, this makes our extension usable for existing work on games and interactive applications.

We have implemented versions of parameterized MSF using both parametric monads (via the `monad-param` library), and effect monads [Orchard and Petricek 2015] (via the `effect-monad`). The implementation of these functions, as well as any kind of type-dependent programming in Haskell with type-level sets, requires the addition of monoidal laws as type constraints to the types of MSF combinators. In the case of `monad-param`, the type system finds it harder to deduce the types of intermediate monads in do-blocks. The use of additional GHC plugins [Bracker and Nilsson 2016] facilitates working with these structures.

The features available in Haskell for dependently typed programming are limited, and mostly enabled via a series of extensions. Our current definitions make use of *AllowAmbiguousTypes*, *DataKinds*, *FlexibleContexts*, *FlexibleInstances*, *GADTs*, *KindSignatures*, *MultiParamTypeClasses*, *NoMonomorphismRestriction*, *RebindableSyntax*, *TypeFamilies* and *TypeOperators*.

Perhaps the biggest limitation in practice is the difficulty in Haskell to prove certain trivial laws about sets to fulfill the aforementioned constraints, such as that the empty set is the identity under set union. Effect monads also include the possibility of constraining the monad, and these constraints are included as invariants in bind and, therefore, in every monadic computation that uses bind. Incorporating these constraints makes signatures more complex. We expect some of these limitations to go away as new features of dependently typed programming are introduced in Haskell and they gain more traction among users. These minor nuances do not impact the process

of describing reactive systems in practice, and the syntax and support available in Haskell for type-level sets make it relatively convenient to work with this proposal.

The inclusion of type definitions for monoidal constraints on the parameters of effect monads, which we also followed in this paper, makes some constructions harder. In particular, we could have used, as the monoid, functions that alter the sets of faults, with function composition and identity. This would give us a general mechanism to perform fault tolerance, by removing faults without having to encapsulate an *MSF* in another to make it "safe". We have experimented with creating this extension, but the arity of the *Effect* type class and the definition of a proper *Unit* type make this approach unsuccessful. We have found similar problems when working with *constrained monads* as provided by the `supermonads` library [Bracker and Nilsson 2016].

Idris. We have implemented a basic prototype of Monadic Stream Functions also in Idris, similarly to the implementation in Haskell. The existence of libraries to work with sets and with probabilities, based on the ones we find in Haskell, make the implementation relatively straightforward.

Idris expects certain functions to be total. Due to our basic type being coinductive and due to the flexibility in the monad, proofs of properties of MSFs and our extensible MSFs would require constraining the monad to be strictly positive. Basic proofs of Arrow laws already exist [Bärenz et al. 2016], and a verified proof of these and other properties remains as future work.

In spite of Idris being a dependently typed language and making it easier to work with values at the type level and to define type functions, similar obstacles were found when trying to define operations that worked on type-level sets of faults. Proofs of some of these laws had to accompany the implementation, and some of the simplest ones (e.g., union with the empty set does not change a set) remain as future work[7]. This is simply a result of our type-level encoding of sets being based on sorted, balanced trees, which makes it harder to write proofs of equality between sets.

10 RELATED WORK

In this paper we have presented a technique to bring fault tolerance information to the types of monadic computations. We have expressed two kinds of fault tolerance information: runtime information and compile-time information. The use of runtime information is more suitable when the information changes over time, or when the type system is not powerful enough to make deductions based on this information. The use of compile-time information is more appropriate for data that does not vary dynamically and that it is simple enough for the compiler to analyze it and manipulate it.

Our work falls in the intersection between Fault Tolerance, Type Systems and Functional Programming. However, many aspects of fault tolerance are not addressed by this paper. For example, we have not discussed techniques to determine which faults to consider, or how to define and delimit Fault Containment Regions, or fault detection and masking algorithms, error recovery, or proofs of consensus and agreement. For an in-depth description of the field, see [Avižienis 1967, 1976; Butler 2008].

Functional Programming. The use of functional languages in critical systems is not new.

OCaml has been used to implement compilers and tools [Pagano et al. 2009] that comply with strict aviation guidelines [199 1992].

In the area of Haskell, specifically, we find that the language has been used both for code generation from embedded DSLs and for verification. Co-pilot [Pike et al. 2010, 2012] is a constrained

[7] Idris defines a keyword, $believe_me$, which allows us to defer proving the most basic statements. Care has been taken to minimize the use of this keyword to only laws that we know from other fields to be true. The framework computes the type-level sets of faults correctly, and lack of these proofs does not invalidate the claims of this paper or the usefulness of this work.

Domain-specific Language to specify runtime monitors that detect property violations on stream-based C applications. The language is implemented as a Haskell DSL whose execution generates correct C code. This effort is complementary to our proposal, as runtime monitors seek to detect faults during execution (as opposed to specifying them in the types and check them during compilation). In theory, it might be possible to combine both frameworks and provide a variant in which external data streams are tagged with the kinds of unhandled faults that may have affected them.

Walker et al. [Walker et al. 2006] present a lambda calculus with fault tolerance information, in which each value is calculated several times and a majority vote is used to discard incorrect values. A similar approach, which carries three values for each result and provides only majority voting as a way to incorporate fault tolerance, might be introduced systematically in our reactive specification by means of a type **data** $FTTriplet\ a = FTTriplet\ (R\ a, G\ a, B\ a)$, where R, G and B are isomorphic to *Identity*, together with a voting function $vote :: Eq\ a \Rightarrow FTTriple\ a \rightarrow a$ that discards up to possibly one different value and uses majority voting to pick one of the other two. Because *FTTriple* forms a monad, it would be immediately usable with our reactive framework. However, there are two strong differences between using this approach and the original proposal by Walker. On the one hand, without a way to indicate to the compiler that it should not carry out common subexpression elimination, the fault tolerance introduced by this approach might be eliminated by the compiler during one of the optimisation stages, an issue that Walker et al. report and deal with explicitly in their system. On the other hand, our proposal is more versatile in the different ways in which fault tolerance can be introduced, allowing us to deal with more complicated fault models and making it more suitable for the controls/engineering domain in which we seek to apply it.

Huch and Norbisrath introduce an extension to Haskell for distributed programming [Huch and Norbisrath 2000]. This extension provides some elementary facilities to tolerate specific kinds of faults in message passing applied to distributed networks. The definition of fault-tolerance in that Huch and Norbirath use is, however, restricted to software implementations, and much narrower than the one presented in [Avižienis 1967]. In particular, it does not consider general fault detection and masking, the definition of fault containment regions, the specification of clear fault models, or the use of type systems to guarantee compliance with a fault model. Cloud Haskell [Epstein et al. 2011] includes primitives to handle fail-stop faults in software processes, similar to the *supervisor* mechanism provided by Erlang [Vinoski 2007], but does not use type-level programming to capture properties of the type model and guarantee that all the faults in the fault model are handled properly. Glasgow Distributed Haskell (GdH) is a Haskell extension for distributed systems [Trinder et al. 2000]. The authors do identify the same key aspects of fault tolerance that we work with, but they only deal with the runtime aspects and present mechanisms to detect errors and recover from them, but not to detect potential faults using the type system. Furthermore, the way that some faults are addressed (re-execution) does not guarantee that the system tolerates that class of faults, and at best transforms fail-stops into delays. Depending on the timing requirements of the system built on top of GdH, long delays might still constitute serious faults. This makes the proposed approach limited in its fault tolerance, and further analysis and a clear fault model are needed. HdpH [Maier et al. 2014; Stewart 2013; Stewart et al. 2016, 2013] is a variant of distributed parallel Haskell for reliable computation. HdpH does provides monitoring and recovering capabilities but, like other proposals for distributed Haskell, its fault tolerance mechanisms are applied during runtime, not during compile time, and it does not provide a mechanism to verify the correct handling of fault classes.

Stream programming in critical systems. Synchronous dataflow programming languages have been used to specify well-formed stream-based mission-critical systems. Esterel [Boussinot and De Simone 1991] is a synchronous language with formally verified semantics that has been used

in avionic software development[Berry et al. 2000]. Scade[Dormoy 2008] is a language and set of tools based on Lustre[Halbwachs et al. 1991] that provides static analysis and verification. These languages are based on an underlying notion of stream programming, which is complementary to our proposal. Just like with Copilot, these synchronous languages could be extended with a notion of explicit type-level fault tolerance.

Monads and Effects. Types like *Maybe*, *Either* and list have been used often to capture failures or exceptions in lazy functional languages [Spivey 1990; Wadler 1985]. The difference between these types and our proposal is that these types assume that failures can be detected, whereas we do not work under that assumption. Nevertheless, there is a class of failures, termed *omissive* or *benign*, defined as those that can be detected as failures, and key in the design of fault-tolerant systems is to add redundancy to make some undetectable or *transmissive* faults detectable or *omissive*. The combination of an exception monad with a fault tolerance monad could provide a very explicit description of all the faults that are not handled, and, of those that are handled, a partitioning between transmissive and omissive failures.

Mechanisms like *extensible effects* [Kiselyov et al. 2013] have been proposed as a composable alternative to monad transformers [Liang et al. 1995b]. Their use and, especially, the possibility of combining different kinds of effects irrespective of their stacking order might ease the specification of mechanisms for fault tolerance at different architectural levels. We have explored the use of effect monads or graded monads [Orchard et al. 2014], polymonads [Bracker and Nilsson 2015], and supermonads [Bracker and Nilsson 2016]. We have not explored the use of unconstrained indexed monads [Katsumata 2014; Orchard and Petricek 2014]. The use of some of these constructs could also help represent MSFs that account for the duration of processing a value, or that contain type-level information of how much history the keep. Alternatives can also be found in dependently-typed languages like Idris, with different proposals to compose effects [Brady 2013].

Formal methods in critical systems. PVS is a dependently typed specification language and an associated theorem prover [Owre et al. 1996, 1992, [n. d.]]. PVS has been used in multiple critical systems and fault tolerance [Rushby [n. d.]]. To name a few areas of application, PVS has been used to verify interactive consistency algorithms that handle Byzantine faults[Lincoln and Rushby [n. d.], 1993] and clock synchronization algorithms [Pfeifer et al. 1999]. In aerospace, an area that is heavily regulated, PVS has been used for fault tolerance in aircraft control systems [Di Vito 1999; Dutertre and Stavridou 1997; RW 1996], and manned and interplanetary spacecrafts [Crow and Di Vito 1998; Di Vito 1996; DiVito and Roberts 1996]. To the best of our knowledge, no work attempts to use PVS's type system to capture unhandled faults or provide a partitioning of the fault space based on the fault tolerance mechanisms of the subsystems used.

Formal verification has also been used to proof properties of seL4 [Klein et al. 2014, 2009], an L4 microkernel implementation that has been verified in Isabelle/HOL. While the goal of seL4 is to provide a kernel that is proven free from programming errors, is has been extended with a fault-tolerant real-time scheduler [Xu et al. 2016] that handles some timing faults that due to hardware or software failures.

11 FUTURE WORK

This work has presented a simple approach at capturing aspects of fault tolerance in reactive and Functional Reactive Programming in Haskell. We have shown how to tag existing reactive transformations with reliability factors, with probabilities and with a sense of the possible unhandled faults that may have affected a result. We have also shown that, with dependently typed programming, we can move some of this information to the type level, and carry out compile-type analysis of the faults that may affect a reactive system. We can incorporate different mechanisms of fault

tolerance, by adding redundancy and using it to detect and correct values. We have demonstrated this approach by averaging the results from multiple sensors, thus increasing the reliability of the results, something that is reflected both at the type and at value level. We have given a simple implementation of an extended reactive framework to handle parameterized monads, which we have implemented also using different monadic extensions in Haskell and in Idris. This extension does not require adaptations to handle fault tolerance, and all the proposals explored in this paper are defined orthogonally in separate monads, which is a key strength of our proposal.

We are currently investigating if it would be possible to use this proposal to describe some of the control systems in small, unmanned aircrafts. While the use of Haskell itself during runtime may not be tolerable due to the lack of real-time guarantees imposed by the garbage collector, compiling a reactive network to a C or another language may produce better results [Pike et al. 2010].

Our use of reliability factors was done at execution level, but an encoding of rational numbers at the type level might make it possible to obtain static information about reliability, and a compile-time understanding of the probabilities of different kinds of failures. The use of a dependently typed language, like Idris, would facilitate operating on rational numbers at the type level, and obtaining compile-time reliability factors. The fault tolerance or recovery mechanisms proposed all work similarly: faults are reversed (i.e., reliability factors are inverted, faults removed from sets, etc.). All the proposals included in this paper can be expressed more generally in terms of non-commutative rings. We expect this to let us generalize the recovery and fault tolerance mechanisms.

We have shown both how to run the system in ideal conditions, and how to use the type-checker to verify that faults are specified. An possible addition would be to extend MSFs with a way of injecting faults [Perez and Nilsson 2017]. This could be additionally improved by the use of a tool like QuickCheck [Claessen and Hughes 2000], that tries to check a property against a series of randomly generated inputs. The combination of several of these techniques, to obtain both fault specifications and confidence factors, remains as future work.

In terms of fault tolerance, we have left aside the study of faults that are related to other faults, as well as timing faults. To a certain extent, dependent probabilities can already be expressed in our proposal. Nevertheless, we expect to explore this further in the future, in combination with ongoing work on MSFs with type-level clocks [Bärenz and Perez 2018] and integration with QuickCheck.

ACKNOWLEDGEMENTS

The author thanks Alwyn Goodloe, Paul Miner, Wilfredo Torres, Kerianne Hobbs, Frank Dedden, and William Edmonson for multiple discussions on fault tolerance. The author also thanks Swee Balachandran, Kerianne Hobbs and Frank Dedden for suggestions and comments on earlier versions of this paper, and anonymous reviewers for their detailed feedback.

This material is based upon work funded under NASA Cooperative Agreement 80LARC17C0004. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views, either expressed or implied, of any of the funding organizations.

REFERENCES

1992. DO-178B: Software Considerations in Airborne Systems and Equipment Certification. (1992).

Algirdas Avižienis. 1967. Design of Fault-tolerant Computers. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference (AFIPS '67 (Fall))*. ACM, New York, NY, USA, 733–743. https://doi.org/10.1145/1465611.1465708

Algirdas Avižienis. 1976. Fault-tolerant systems. *IEEE Trans. Computers* 25, 12 (1976), 1304–1312.

Manuel Bärenz and Ivan Perez. 2018. Rhine - FRP with type-level clocks. In *Haskell Symposium*. (Submitted).

Manuel Bärenz, Ivan Perez, and Henrik Nilsson. 2016. Mathematical Properties of Monadic Stream Functions. http://cs.nott.ac.uk/~ixp/papers/msfmathprops.pdf. (2016).

Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert De Simone. 2000. Esterel: A formal method applied to avionic software development. *Science of Computer Programming* 36, 1 (2000), 5–25.

Frédéric Boussinot and Robert De Simone. 1991. The ESTEREL language. *Proc. IEEE* 79, 9 (1991), 1293–1304.

Jan Bracker and Henrik Nilsson. 2015. Polymonad Programming in Haskell. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages (IFL '15)*. ACM, New York, NY, USA, Article 3, 12 pages. https://doi.org/10.1145/2897336.2897340

Jan Bracker and Henrik Nilsson. 2016. Supermonads: One Notion to Bind Them All. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 158–169. https://doi.org/10.1145/2976002.2976012

Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 133–144.

Ricky W. Butler. 2008. *A Primer on Architectural Level Fault Tolerance.* Technical Report NASA/TM-2008-215108, L-19403. NASA Langley Research Center.

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa arcade. In *Haskell Workshop*. 7–18.

Judith Crow and Ben Di Vito. 1998. Formalizing Space Shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 3 (1998), 296–332.

Ben L Di Vito. 1996. Formalizing new navigation requirements for NASA's space shuttle. In *International Symposium of Formal Methods Europe*. Springer, 160–178.

Ben L Di Vito. 1999. A model of cooperative noninterference for integrated modular avionics. In *Dependable Computing for Critical Applications 7, 1999*. IEEE, 269–286.

Ben L DiVito and Larry W Roberts. 1996. Using formal methods to assist in the requirements analysis of the space shuttle GPS change request. (1996).

Francois-Xavier Dormoy. 2008. Scade 6: a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTSâĂŹ08)*. 1–9.

Bruno Dutertre and Victoria Stavridou. 1997. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering* 23, 5 (1997), 267–278.

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *ACM SIGPLAN Notices*, Vol. 32(8). 263–273.

Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. 2011. Towards Haskell in the cloud. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 118–129.

Martin Erwig and Steve Kollmansberger. 2006. Functional pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming* 16, 1 (2006), 21–34.

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

Frank Huch and Ulrich Norbisrath. 2000. Distributed programming in Haskell with ports. In *Symposium on Implementation and Application of Functional Languages*. Springer, 107–121.

John Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming* 37, 1 (2000), 67 – 111.

Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. *ACM SIGPLAN Notices* 49, 1 (2014), 633–645.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 59–70.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)* 32, 1 (2014), 2.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

Sheng Liang, Paul Hudak, and Mark Jones. 1995a. Monad transformers and modular interpreters. In *Symposium on Principles of programming languages*. 333–343.

Sheng Liang, Paul Hudak, and Mark Jones. 1995b. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 333–343.

Patrick Lincoln and John Rushby. [n. d.]. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *Computer Assurance, 1994. COMPASS'94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*. IEEE, 107–120.

Patrick Lincoln and John Rushby. 1993. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*. IEEE, 402–411.

Patrick Maier, Robert Stewart, and Phil Trinder. 2014. The HdpH DSLs for Scalable Reliable Computation. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2633357.2633363

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Haskell Workshop*. 51–64.

Dominic Orchard and Tomas Petricek. 2014. Embedding Effect Systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2633357.2633368

Dominic Orchard and Tomas Petricek. 2015. Embedding effect systems in Haskell. *ACM SIGPLAN Notices* 49, 12 (2015), 13–24.

Dominic A. Orchard, Tomas Petricek, and Alan Mycroft. 2014. The semantic marriage of monads and effects. *CoRR* abs/1401.5391 (2014). arXiv:1401.5391 http://arxiv.org/abs/1401.5391

Sam Owre, Sreeranga Rajan, John M Rushby, Natarajan Shankar, and Mandayam Srivas. 1996. PVS: Combining specification, proof checking, and model checking. In *International Conference on Computer Aided Verification*. Springer, 411–414.

S. Owre, J. M. Rushby, , and N. Shankar. 1992. PVS: A Prototype Verification System. In *11th International Conference on Automated Deduction (CADE) (Lecture Notes in Artificial Intelligence)*, Deepak Kapur (Ed.), Vol. 607. Springer-Verlag, Saratoga, NY, 748–752. http://www.csl.sri.com/papers/cade92-pvs/

Sam Owre, Natarajan Shankar, John M Rushby, and David WJ Stringer-Calvert. [n. d.]. PVS language reference. ([n. d.]).

Bruno Pagano, Olivier Andrieu, Thomas Moniot, Benjamin Canou, Emmanuel Chailloux, Philippe Wang, Pascal Manoury, and Jean-Louis Colaço. 2009. Experience Report: Using Objective Caml to Develop Safety-critical Embedded Tools in a Certification Framework. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 215–220. https://doi.org/10.1145/1596550.1596582

Ivan Perez. 2017. Back to the Future: Time Travel in FRP. In *Proceedings of the 10th ACM SIGPLAN International Haskell Symposium (Haskell 2017)*. ACM, New York, NY, USA, 12. https://doi.org/10.1145/3122955.3122957

Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional Reactive Programming, Refactored. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. ACM, New York, NY, USA, 33–44. https://doi.org/10.1145/2976002.2976010

Ivan Perez and Henrik Nilsson. 2017. Testing and Debugging Functional Reactive Programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 2 (Aug. 2017), 27 pages. https://doi.org/10.1145/3110246

Holger Pfeifer, Detlef Schwier, and Friedrich W Von Henke. 1999. Formal verification for time-triggered clock synchronization. In *Dependable Computing for Critical Applications 7, 1999*. IEEE, 207–226.

Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. 2010. Copilot: a hard real-time runtime monitor. In *International Conference on Runtime Verification*. Springer, 345–359.

Lee Pike, Sebastian Niller, and Nis Wegmann. 2012. Runtime Verification for Ultra-Critical Systems. In *Runtime Verification*, Sarfraz Khurshid and Koushik Sen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 310–324.

John Rushby. [n. d.]. PVS Bibliography. http://pvs.csl.sri.com/documentation.shtml. ([n. d.]).

Butler RW. 1996. An introduction to requirements capture using PVS: Specification of a simple autopilot. (1996).

Mike Spivey. 1990. A functional theory of exceptions. *Science of computer programming* 14, 1 (1990), 25–42.

Robert Stewart. 2013. *Reliable massively parallel symbolic computing: fault tolerance for a distributed Haskell*. Ph.D. Dissertation. Heriot-Watt University.

Robert Stewart, Patrick Maier, and Phil Trinder. 2016. Transparent fault tolerance for scalable functional computation. *Journal of Functional Programming* 26 (2016).

Robert Stewart, Phil Trinder, and Patrick Maier. 2013. Supervised Workpools for Reliable Massively Parallel Computing. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 247–262.

Phil Trinder, Robert Pointon, and Hans-Wolfgang Loidl. 2000. Towards Runtime System Level Fault Tolerance for a Distributed Functional Language. *Trends in Functional Programming* 2 (2000), 103.

Steve Vinoski. 2007. Reliability with Erlang. *IEEE Internet Computing* 11, 6 (2007).

Philip Wadler. 1985. How to replace failure by a list of successes. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 113–128.

David Walker, Lester Mackey, Jay Ligatti, George A. Reis, and David I. August. 2006. Static Typing for a Faulty Lambda Calculus. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 38–49.

Libin Xu, Yuebin Bai, Kun Cheng, Lingyu Ge, Danning Nie, Lijun Zhang, and Wenjia Liu. 2016. Towards Fault-Tolerant Real-Time Scheduling in the seL4 Microkernel. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*. IEEE, 711–718.