ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

A Functional Reactive Perspective on the Aggregate Computing Paradigm

Tesi di laurea in
PERVASIVE COMPUTING

Relatore
Prof. Mirko Viroli

Correlatore
Dott. Roberto Casadei
Dott. Gianluca Aguzzi

Candidato
Francesco Dente

IV Sessione di Laurea
Anno Accademico 2021-2022

Abstract

The challenges faced by developers when dealing with large-scale, distributed systems call for high level abstractions in order to manage their growing complexity. In this context, aggregate computing is an emerging paradigm that allows to declaratively specify the behavior of these types of systems, by viewing the system as a whole, rather than focussing on the interactions of individual devices. This new way of reasoning about the behavior of aggregates of devices is based on a formal calculus that describes programs as the functional composition of computational fields that evolve through time, called "field calculus". As it currently stands, though, this formal model does not provide native ways to fine-tune the timing and the evolution dynamics of fields, which is desirable to avoid wasteful usage of processing resources.

In this paper we propose to combine aggregate computing with the functional reactive programming to develop an evolution of the current model proposed by field calculus that allows to effectively specify aggregate computations as reactions to changes in the environment. A small prototype has been developed to assess the applicability of the proposed model and to verify that some of the properties of the original model remain observable in the new one. Ultimately, this project aims to contribute towards the vision of functional reactive self-organization.

Contents

List of Figures

Chapter 1

Introduction

The complexities that developers face nowadays when dealing with large-scale, distributed systems have grown to a point that imposes the adoption of new techniques to manage those complexities and to help reasoning about distributed software at high levels of abstraction. Typically, systems of this nature fall under the category of *Collective Adaptive Systems*, in which a large number of devices pursues a global goal by means of *strongly decentralized* interactions, while adapting their behavior to the constantly changing environment. New paradigms should allow to *declaratively* specify the behavior of such types of system, while also providing *composable* and *reusable* building blocks.

One of the most promising approaches in this matter is *aggregate computing* [12], which focuses on the definition of behaviors of aggregates of devices, rather than thinking in terms of single entities. This leads to the idea of *self-organization* [11], thanks to which global coordination behavior *emerges* from local coordination abstractions. Its foundation is given by *field calculus*, a formal calculus that defines aggregate programs as the functional composition of *computational fields*.

A set of frameworks and languages based on this paradigm already exist, both as internal or external Domain Specific Languages, like ScaFi or Protelis. However, these frameworks – and field calculus as well – propose a *proactive model* based on computation rounds, which lacks abstractions that naturally model the behavior of a system as a reaction to relevant events in the surrounding environment. This means that it is not currently possible to fine-tune the overall dynamics and the timing of computations, resulting in wasteful usage of the processing resources.

The objective of this thesis is to explore the potential problems connected with the current proactive model of field calculus, proposing a new one based on *reactivity* to environmental changes, leveraging the principles of *functional reactive programming* [2] to keep the compositionality aspects intact and to improve on some of the current shortcomings. On top of that, a small, but functional prototype will be developed for said model, both to assess its feasibility and to be aware of some of the caveats that a production-ready implementation should tackle. Ultimately, the vision of the whole project is to take a step towards *functional reactive self-organization*.

Thesis structure The remainder of this thesis is organized as follows. Chapter 2 introduces the essential background on all the subjects upon which all the work is based, presenting concepts of *functional programming*, the new features of *Scala 3*, the basics of *aggregate computing* and *field calculus*, and finally an overview of the *functional reactive programming* paradigm. On this foundation, Chapter 3 analyzes the current state of aggregate computing, in particular with the *ScaFi* framework, and proposes a specification for a reactive model that . Subsequently, Chapter 4 goes into the details of how the prototype for the specification was implemented with Scala 3. Chapter 5 shows the process that was adopted in order to evaluate both the model and the implementation, through unit tests and empirical tests. Finally, Chapter 6 gives some final thoughts on the contribution of the thesis and paves the way for future work.

Chapter 2

Background

This chapter gives an high level overview of the concepts, paradigms, and frameworks that were used as references throughout the development of this thesis.

2.1 Functional programming

Functional Programming is a programming paradigm in which computation is expressed as a transformation of inputs into outputs using functions and function composition. Here, functions are to be intended in the *mathematical* sense of the term, meaning that they are just mappings from elements of their domain to elements of their co-domain.

2.1.1 Functional programming concepts

This section illustrates some of the key concepts on which functional programming is based upon and that are key when working in a functional style.

Purity The mathematical definition of a function can be adapted to the context of computer programming, in which mathematical functions are referred to as **pure functions**. To be pure, a function must satisfy the following properties:

- given the same arguments, it always returns the same value, or in other words, its output depends solely on its arguments;

- it must produce no observable side effects (e.g., mutate global shared state, send data over output streams, etc).

Referential transparency A sub-expression (i.e., the application of a function) is said to be *referentially transparent* if it can be safely substituted with its final value without changing the overall semantics of the complete expression that contains it. This is a direct consequence of function purity, since a pure function's only observable effect is the value it returns for its given arguments and that value is consistent over time. Due to this fact, referential transparency and purity are often used (erroneously) as synonyms. While it is true that purity implies referential transparency, the opposite is not, in general.

Evaluation order strategies Functional languages typically support different ways to evaluate arguments as they are passed to functions. There are three main approaches:

- *call-by-value*: arguments are evaluated before the function application;
- *call-by-name*: arguments are evaluated each time they're value is required inside the body of the function;
- *call-by-need*: also referred to as *lazy* evaluation, arguments are evaluated once the first time their value is needed inside the body of the function.

Immutability One of the consequences of purity when writing code in a functional style is that programs are not allowed to have any global state that is shared among their sub-components. This also means that the data structures to be used must be designed accordingly and should therefore be *immutable*, allowing state changes only via creating new versions of the same data structure. This is typically not a problem, since complex data structures (e.g., collections like lists, sets, maps, ...) are optimized to reuse as much as possible when creating copies of themselves.

Higher order functions Functional programming proposes the idea of *functions as first-class citizens*, indicating that functions can also be considered values and be passed around just like traditional values. Therefore a function can take functions as arguments or return functions as outputs, in which case is called an *higher order function*.

2.2 Scala 3

Scala is a programming language built on top of the Java Virtual Machine (JVM), with support for both the *object-oriented* and the *functional* paradigms [6], effectively making it a *multi-paradigm language*. It is a pure object-oriented language, in fact everything is an *object* and objects are defined through *classes* and *traits*. It is a functional language, because it supports the idea of functions as values and allows higher order functions to be defined [9].

The main features of Scala are:

- support for both OOP and FP;
- seamless integration with the Java ecosystem and the JVM;
- powerful static type system;
- flexible syntax that makes it a scalable language;

The most recent version of Scala (at the time of writing) is Scala 3 [10], which is the version that has been used during development. Scala 3 introduces several new constructs and improvements over its predecessor [5], that were considered to be necessary to streamline the development process and allow for a concise and conveying syntax. The most relevant that were also used for this project can be summarized as follows:

- **optional braces** introduce the ability to use a python-like syntax that controls scopes through indentation;
- **enumerations** improve on the traditional way of defining *algebraic data types* (see Section 2.2.1) that required the use of `sealed traits`;
- **improved contextual abstractions** replace the use of the old `implicits` with new constructs that better convey the developer's intent, in particular:
  - `using` clauses allow methods to abstract over information that is available in the calling context and that should be passed implicitly (see Section 2.2.2);
  - `given` instances define canonical values for types that can be passed implicitly (see Section 2.2.2) and are typically used to define *type class instances*;
  - `extension` methods allow developers to attach methods to types after their definition (see Section 2.2.3);
  - implicit conversions let the compiler view a type as another and perform conversions without explicit casting;
  - context functions define types of functions that only take context parameters (see Section 2.2.4);

The following sections give more details on these constructs.

2.2.1 Enumerations and algebraic data types

An algebraic data type (ADT) is a structured type defined as a composition of other types, which can in turn be other ADTs or atomic types. ADTs can belong to one of these categories:

- **Product Types** (also known as *records*): defined as the cartesian product of its composing types;
- **Sum Types** (also known as *discriminated unions*): defined as the union of its composing types.

As an example, let's consider modeling a binary tree. A binary tree can be represented as an ADT through a sum type `Tree`, which is a discriminated union of these product types:

- `Leaf(x)`, where `x` is the value contained in the leaf itself;
- `Node(x, left, right)` where x is the value contained in the node itself, and `left` and `right` are respectively the left and right subtrees.

This ADT indicates that a binary tree is either a *leaf*, in which case it only stores the value it contains, or a *node*, in which case it stores its value a the left and right subtrees. Note that this is a recursive definition, since `left` and `right` are also of type `Tree`.

In Scala 2, an ADT could be defined using `sealed traits` (to represent sum types) in combination with `case classes` (modeling product types). Note the use of the `sealed` keyword, which guarantees that a trait decorated with this modifier will not be extended in other source files other than the one where the trait is declared. This is fundamental, since a discriminated union should not be open to extension and allow only its composing types to be used. In addition, `case classes` are a perfect candidate to represent product types due to their structure (resembling that of records) and can be used seamlessly with *pattern matching*.

In Scala 2, the binary tree example would be implemented as follows:

```scala
sealed trait Tree[T]
case class Leaf[T](x: T) extends Tree[T]
case class Node[T](x: T, left: Tree[T], right: Tree[T]) extends Tree[T]
```

Scala 3 takes an approach to ADTs that better conveys the intent of the developer, offering a structure that is specifically designed to represent discriminated unions. The new version introduces the `enum` keyword, which smooths out the process of defining discriminated unions. The Scala 3 version of the binary tree example becomes

```
enum Tree[T]:
  case Leaf(x: T)
  case Node(x: T, left: Tree[T], right: Tree[T])
```

which is more succinct and avoids the boilerplate required by its predecessor.

Finally, both approaches are well suited to be used with pattern matching (although only Scala 3 is shown here). This example show the implementation of a method that converts a binary tree to a string, by using *preorder traversal*.

```
import Tree._

def stringify[T](tree: Tree[T]): String = tree match
  case Leaf(x) => x.toString
  case Node(x, l, r) => s"$x[${stringify(l)},${stringify(r)}]"

val aTree = Node(
  1,
  Node(
    2,
    Leaf(3),
    Leaf(4)
  ),
  Leaf(5)
)

println(stringify(aTree)) // 1[2[3,4],5]
```

2.2.2 Using clauses and given instances

The `using` and `given` keywords were introduced in Scala 3 as a replacement for the `implicit` keyword when passing *context parameters* and when defining *canonical values*. Before Scala 3, in fact, the `implicit` mechanism lacked on clarity of intent, since the keyword was also used to define extension methods (see Section 2.2.3) and implicit conversions.

The following snippet shows how `implicits` worked before Scala 3. First, a developer would need a method accepting a parameter `implicitly`. Here, the `printBoth()` method accepts a `printer` argument that should be inferred from the caller context and not be passed explicitly.

```
trait Printer {
  def print(x: Any): Unit
}

def printBoth(x: Any, y: Any)(implicit printer: Printer): Unit = {
  printer.print(x)
  printer.print(y)
}
```

Next, when calling the `printBoth()` method, Scala tries to synthesize an appropriate value of type `Printer` that is marked as `implicit` in the calling scope:

```scala
implicit val printer: Printer = x => println(x)

// The following lines are equivalent.
printBoth("A", "B")
printBoth("A", "B")(printer)
```

Note that the `printer` value is declared using the `implicit` keyword, marking it as suitable to be passed implicitly.

As stated at the beginning of this section, Scala 3 replaces the `implicit` keyword with `using` and `given`. Other than that, not much changes in the structure of the previous code:

```scala
trait Printer:
  def print(x: Any): Unit

def printBoth(x: Any, y: Any)(using printer: Printer): Unit =
  printer.print(x)
  printer.print(y)

...

given printer: Printer = x => println(x)

// The following lines are equivalent.
printBoth("A", "B")
printBoth("A", "B")(using printer)
```

Note that, just like `implicit` parameters, `using` parameters are available as `given`s inside the scope of the method or function they are declared in. This makes the following snippet valid Scala code:

```scala
def nPrintBoth(x: Any, y: Any)(n: Int)(using Printer): Unit =
  (1 to n).foreach(_ => printBoth(x, y))
```

Also note that Scala 3 lets the developer omit `using` parameters names, since most of the times they are in turn passed downstream implicitly. In those cases, they are accessible through the `summon[T]` method, which finds the given instance of type `T` available in the current context.

2.2.3 Extension methods

Scala 3 introduced extension methods following the same principles that inspired the `given`/`using` keywords, that is the overloaded meaning of the `implicit` keyword. In fact, prior to this version, extension methods were also written using that keyword, in particular by using `implicit classes`. The following listing shows how it was done:

```scala
object Extensions {
  implicit class RichInt(x: Int) {
    def isDivisibleBy(other: Int): Boolean = x % other == 0
  }
}

import Extensions._
println(10.isDivisibleBy(5))
```

Other than the use of an overloaded keyword, this approach had the disadvantage of having to come up with a name for the implicit class, that would however never be referenced again afterwards.

Scala 3 solved this issue by adding the `extension` keyword. The previous code, re-written in Scala 3, would be as follows:

```scala
object Extensions:
  extension (x: Int)
    def isDivisibleBy(other: Int): Boolean = x % other == 0

import Extensions._
println(10.isDivisibleBy(5))
```

2.2.4 Context functions

Context functions are a completely new feature of Scala 3, allowing developers to create function types that only accept *context parameters.*

A context function type is written as

`(T1, ..., Tn) ?=> U`

which represents a function accepting `n` context arguments with types `(T1, ..., Tn)` and returning a value of type `U`. Context functions can be used just as normal functions, but they have special syntax for passing and getting arguments implicitly.

Consider the following snippet:

```scala
given world: String = "World"

def greet(how: String ?=> String): Unit = println(how)
// def greet(how: String ?=> String): Unit = println(how(using world))

// The following lines are equivalent.
greet(s"Hello ${summon[String]}")
greet(who ?=> s"Hello $who")
```

Here `greet()` is a method accepting a context function whose context parameter is of type `String` and also returns a `String`. Note that the implementation of `greet()` invokes `how` by letting the compiler synthesize its argument implicitly, using the `given String` declared at the top of the snippet. Moreover, both invocations of `greet()` are equivalent since Scala expands the first expression to a context function literal behind the scenes.

In general, if an expression `E` is expected to have a context function type `(T1, ..., Tn) ?=> U` and it is not already a context function literal, it is re-written as `(x1: T1, ..., xn: Tn) ?=> E` automatically by the compiler. Also, while type-checking `E`, its arguments are available as givens, which means that they are accessible using `summon[T]`.

2.3 Aggregate computing

Aggregate computing is an emerging approach to the engineering of complex coordination for distributed systems, in particular *Collective Adaptive Systems* [12]. It is mostly based on the idea that it is simpler to view system interactions in terms of information propagating through collectives of devices, rather than in terms of individual devices and their interaction with their peers and environment.

Aggregate computing fits particularly well when the nature of the problem requires to deal with a *network of devices* with these features:

- **openness**, meaning that the environment in which devices are immersed can exhibit unexpected changes and faults;
- **large scale**, with a possibly huge number of devices/agents that need to coordinate and require good abstractions to be managed;
- **intrinsic adaptiveness**, that is, the ability to react to relevant events in order to guarantee overall system resilience.

These concerns call for an approach based on *self-organization*, where global and robust coordination behavior emerges from local coordination abstractions.

Another goal of aggregate computing is to give developers a way to describe the behavior of distributed systems with the features described above in a *composable* and *declarative* fashion, in order to scale well with the complexity of the domain. This is possible thanks to the *mathematical core* of aggregate computing, that is based on *computational fields* (see Section 2.3.1) and *field calculus* (see Section 2.3.2).

2.3.1 Basic abstractions

Aggregate computing models a distributed system as a set $\mathcal{D}$ of devices, ranged over by δ. On top of that, a reflexive [1] *neighboring relation* indicates the devices with which one can communicate (which is application-dependent, and can be used to describe logical or physical proximity). In addition, each device has a set of *sensors* that enable the perception of the environment.

[1]each device is a neighbor of itself.

The primary abstraction introduced by aggregate computing is the *computational field* (or simply *field*), which is a function $\phi : \mathcal{D} \mapsto \mathcal{L}$ mapping each device $\delta \in \mathcal{D}$ to a local value $l \in \mathcal{L}$ [11]. A *field evolution* is a dynamically changing field, and a *field computation* takes field evolutions as inputs and produces field evolutions as outputs. These outputs are defined in such a way that they change tracking input changes.

The key idea of aggregate computing is that any field computation (*global interpretation*) can be mapped to a *single-device behavior* that is iteratively executed by all the devices in the network (*local interpretation*). Each iteration executed by a device is called a *computation round* and can be subdivided in three steps:

- **sense**: the device gathers information coming from its neighbors and local sensors, which are collected to create its *local context* (or *local state*) for the current round;
- **eval**: the computation defined by the behavior is evaluated against the local context, producing an *export*;
- **broadcast**: the export is broadcast to all the device's neighbors, which in turn collect and use this information in their own future rounds.

2.3.2 Field calculus

Field calculus represents a simple language that can be used to describe computations acting on fields. While its syntax, typing, and semantics are deeply discussed in [12] and are omitted here for simplicity, a brief description of its elements is presented below:

- a *field calculus program* $\mathtt{P}$ consists of a sequence of *function declarations* $\bar{\mathtt{F}}$ followed by the *main expression* $\mathtt{e}$;
- an expression $\mathtt{e}$ can be:
  - a *variable* $\mathtt{x}$ (e.g., function parameters, ...);
  - a *local value* l (e.g., booleans, numbers, strings, tuples, ...);
  - a *neighboring field value* ϕ, i.e., a (per-device) mapping from the devices's neighbors to local values;
  - a *function call* $\mathtt{f}(\bar{\mathtt{e}})$ to a *user-declared function* or a *built-in function* (e.g., maths or boolean operators, sensors, ...);
  - a *branching expression* $\mathtt{if}(\mathtt{e}_1)\{\mathtt{e}_2\}\{\mathtt{e}_3\}$ which splits computation into isolated sub-regions, resulting in $\mathtt{e}_2$ where and when $\mathtt{e}_1$ evaluates to true, and in $\mathtt{e}_3$ otherwise;

- a `nbr(e)` construct, which creates a neighboring field value that maps each neighbor to the latest result of evaluating `e`;
- a `rep(e`$_1$`){(x)` $\Rightarrow$ `e`$_2$`}` construct, which models state evolution over time.

To work properly, the semantics of `nbr` and `rep` require a way to access, respectively, the last registered state of each neighbor and the last registered output of the device itself. In addition, this process should be made in such a way that different instances of `rep` and `nbr` cannot inadvertently "swap" their respective value. This process is called *alignment*, and it has the consequence that two branches of an `if` expression execute in isolation, meaning that two devices that execute different branches cannot communicate with each other inside their branches. In practice, this process is done by carefully constructing the export of an expression as an *evaluation tree* that represents the aggregate computation. The complete semantics of export construction and alignment can be found in Appendix C of [11].

2.4 Functional reactive programming

Functional Reactive Programming (FRP) is a style of programming that unifies the approaches of *Reactive Programming* (RP) and *Functional Programming* (FP, see Section 2.1) to tame the complexity of *event-driven interactive applications*.

The following sections describe the conceptual evolution of the paradigms used to deal with event-driven scenarios, starting from the traditional *observer pattern* and evolving into *FRP*. The discussion is primarily based on the book [2] and on the **Sodium** library written by the authors of the book (see Section 2.4.3 for further details).

2.4.1 Limitations of the observer pattern

The observer pattern is a way to define event-driven logic by registering *callbacks* (or *listeners*) to the sources of events, creating a one-to-many dependency from the producer of events to its consumers. This is the traditional (and mainstream) approach to dealing with stateful, event-driven logic, which is typically associated with applications like GUIs or games.

Typically, the observer pattern is associated to some *state machine* that implements the *domain logic* while encapsulating the application *state*. This model can be described as follows:

- an event gets pushed to the listener;

- the listener activates the logic based on the input event and on the current state;
- the logic updates the current state;
- the logic may produce some output that is not fed back into the state.

While the observer pattern can be pretty straightforward for simple enough scenarios, the growing complexity of modern event-driven applications brings to light some of its shortcomings, which are called by the book "the six plagues of listeners":

- *unpredictable order*: since listeners are typically notified in the order they are registered in, keeping track of the temporal dependencies between listeners can become infeasible and lead to unpredictable behavior;
- *missed first event*: it can be difficult to guarantee that listeners are registered before the first event is sent;
- *messy state*: traditional state machines are hard to reason about and tend to quickly grow in complexity;
- *threading issues*: using locks inside listeners to guarantee thread safety can lead to deadlocks which are difficult to track down;
- *leaking callbacks*: forgetting to unregister a listener can cause memory leaks;
- *accidental recursion*: the order in which state is updated and listeners are notified is crucial and needs careful attention.

2.4.2 FRP paradigm

In order to overcome the limitations of the observer pattern, FRP proposes a paradigm shift built around the notion of *continuous time-varying values* and *propagation of change* [1]. This shift allows for a more *declarative* way of writing programs, since developers only have to describe them in terms of what they do and let the underlying execution model manage when changes should be propagated. In fact, RP promotes the idea of a *dependency graph of values and computation*, that dictates the flow of data and the propagation of changes occurring to each node of the graph.

According to the terminology used in Sodium (see Section 2.4.3), the dependency graph is composed of:

- *cells*, representing values that change over time (possibly in a continuous fashion);

- *streams*, representing sequences of discrete events.

Cells are typically used to represent state and its evolution over time, while streams encode the occurrence of events of interest upon which state should change or actions should be taken.

FRP provides operators to transform and combine cells and streams and obtain new ones, mainly by *mapping*, *filtering*, *merging* or *converting one into the other*. Since there is no standard specification of what these operators should be, they are best explained by referring to Sodium's API and underlying model in Section 2.4.3. These operators are generally used upon startup to construct the dependency graph. This is known as the *initialization* stage. After initialization, the FRP engine takes care of taking inputs and converting it into outputs that act upon external consumers (*running* stage).

2.4.3 FRP in Sodium

Sodium [2] is a library designed to work with FRP and written by the authors of [2]. Despite being mainly developed for Java, it also has a variety of adaptations for other languages, including C#, F# and Scala. However, these adaptations do not always implement all the features that are included in the Java version. For this reason, the remainder of this section presents code snippets that are written in Java.

As stated before, Sodium is primarily based on two types:

- `Cell<T>` represents a value of type `T` that changes over time;
- `Stream<T>` represents a sequence of emissions of events, each holding data of type `T`.

In addition, Sodium implements a series of *primitives* that can be used to perform transformations on cells and streams, ultimately defining the domain logic.

The following sections give an overview of the most relevant features of Sodium.

Core Primitives

Sodium is based on a set of *ten primitives* that form its *conceptual core*. These primitives are where the functional part of FRP comes in. To guarantee the compositionality [3] of FRP, all functions passed to the primitives must be referentially transparent.

The ten primitives are described in more detail in the following paragraphs.

[2] `https://github.com/SodiumFRP/sodium`

[3] The property stating that the meaning of an expression is determined by the meanings of its parts and the rules used to combine them. More practically, compositionality gives some guarantees that working subparts will still work when combined together.

Never Produces a stream that will never emit any events.

```
Stream<String> never = new Stream<>();
```

Sodium does not provide a specific `never` method, but uses the empty constructor of the `Stream<T>` class as a way to create the *never* stream.

Constant Produces a constant cell that will always have the given value.

```
Cell<String> helloWorld = new Cell<>("Hello World!");
```

Just like *never*, there is no method named *constant*, but the constructor of `Cell<T>` accepting a `T` creates a constant cell.

Map (stream) Produces a stream emitting the events emitted by the source stream after applying a mapping function.

```
Stream<Integer> source = ...;
Stream<String> out = source.map(x -> Integer.toString(x));
```

Map (cell) Produces a cell whose value is taken from the source cell by applying a mapping function.

```
Cell<Integer> source = ...;
Cell<String> out = source.map(x -> Integer.toString(x));
```

Merge Combines two streams of the same type together, returning a stream that emits events from either input. Since Sodium supports *simultaneous events* (see Section 2.4.3 for more details), if both input streams happen to emit at the same time the given combining function will be used to produce the final output event.

```
Stream<Integer> left = ...;
Stream<Integer> right = ...;
Stream<Integer> merged = left.merge(right, (l, r) -> l + r);
```

Hold Converts a stream into a cell in such a way that the cell's value is that of the most recent event received. The initial value for the cell (before the first event is emitted by the stream) is the argument passed to `hold()`.

```
Stream<Integer> events = ...;
Cell<Integer> hold = events.hold(0);
```

Snapshot Captures the value of the given cell whenever the source stream fires. Produces a stream firing at the same time as the source stream and emitting combinations of the values from the stream and the cell using the supplied function.

```java
Stream<String> trigger = ...;
Cell<Integer> state = ...;
Stream<String> out = trigger.snapshot(state, (t, s) -> t + s);
```

Filter Produces a stream that emits the events from the source stream only if the pass the given predicate.

```java
Stream<Integer> events = ...;
Stream<Integer> out = events.filter(x -> x > 0);
```

Lift Combines two or more cells into one by combining the values of all input cells using a combining function.

```java
Cell<Integer> left = ...;
Cell<Integer> right = ...;
Cell<Integer> out = left.lift(right, (l, r) -> l + r);
```

Sample Returns the current value of the cell it is invoked on.

```java
Cell<String> state = ...;
String currentState = state.sample();
```

Switch (stream) Flattens a cell of streams into a single stream that emits whenever the active stream for the cell emits.

```java
Cell<Stream<Integer>> source = ...;
Stream<Integer> out = Cell.switchS(source);
```

Switch (cell) Flattens a cell of cells into a single cell whose value is the value of the active cell of the wrapper cell.

```java
Cell<Cell<Integer>> source = ...;
Cell<Integer> out = Cell.switchC(source);
```

External interfacing

The *pure core* of Sodium is not enough to deal with all the requirements of an event-driven application. In fact, these requirements include I/O (e.g., network communication, interfacing with the file system, ...), GUIs, and other concerns that are typically not integrated with Sodium. This requires a way for developers to interface the pure core with the *impure* nature of external concerns, namely:

- *pushing events into streams and cells*;
- *listening to events from streams and cells.*

These somewhat resemble the mechanisms used by the observer pattern (recall Section 2.4.1), but they have special constraints and rules to keep compositionality intact, especially when working with transactions (see Section 2.4.3).

Pushing events into streams and cells Sodium comes with subclasses of `Stream<T>` and `Cell<T>` that expose a `send()` method allowing events and values to be pushed into the FRP logic. These are, respectively, `StreamSink<T>` and `CellSink<T>`. The idea is that a module that deals with an external concern would use these sinks internally to push data into the FRP logic, and only expose those as their "pure" counterparts.

The following listing shows an example of how to turn a Swing `JButton` into a stream of its clicks.

```java
public Stream<Unit> clicks(JButton button) {
    StreamSink<Unit> clicksSink = new StreamSink<>();
    button.addActionListener(e -> clicksSink.send(Unit.UNIT));
    return clicksSink;
}
```

Listening to events from streams and cells The values of streams and cells can be made available to the rest of the application by *listening* to their updates. To do this, `Stream<T>` and `Cell<T>` have a `listen()` method to register a listener that gets notified whenever that stream fires or the value of the cell is updated.

```java
public <X> void printAll(Stream<X> stream) {
    stream.listen(x -> System.out.println("Emitted: " + x));
}
```

Transactions

Most FRP systems support the idea that multiple events can be *simultaneous*. This can be done by defining a boundary (which is called a *transaction* in Sodium) in which every event is considered to be happening at the same time. Naturally, FRP primitive should be designed to take this aspect into account, and provide a way to handle simultaneous events correctly (recall the *merge* primitive for example). All Sodium primitives are automatically run inside transactions, but developers can manually define transactions using `Transaction.run(() -> ...)` or `Transaction.runVoid(() -> ...)`.

Transactions can be used primarily for two reasons:

- wrapping the construction of the dependency graph in a single instance to avoid the *missed first event* plague;
- using `Sinks` to send multiple events that should be considered simultaneous.

Looping streams and cells

In FRP there's often the need to have the definition of a cell or of a stream depend on itself. Since programming languages (like Java in this case) do not typically allow references to variables before their declaration, Sodium provides a mechanism that simulates *forward references* [4]. These are `CellLoop<T>` and `StreamLoop<T>`. The idea is that these classes are, respectively, subclasses of `Cell<T>` and `Stream<T>` that can act as placeholders for cells and streams before their initialization, and that can be freely used as their counterparts. After all the dependencies have been setup and the real value for the placeholders can be initialized, this can be done by calling the `loop()` method.

An example where looping is necessary could be an accumulator, which consists of a cell whose value is the sum over time of values emitted by a stream.

```java
public Cell<Integer> accumulate(Stream<Integer> deltas) {
    return Transaction.run(() -> {
        CellLoop<Integer> output = new CellLoop<>();
        Stream<Integer> updates = deltas.snapshot(output, (c, d) -> c + d);
        output.loop(updates.hold(0));
        return output;
    });
}
```

It is worth noting that looping will fail with an exception if it is not wrapped inside a transaction, in order to avoid updates coming before the call to `loop()`.

[4]Referencing an identifier that has not been declared yet.

Chapter 3

Analysis and Design

Leveraging the notions coming from Chapter 2, this chapter inspects the current state of aggregate computing, identifying some of the missing features of the current model as proposed by field calculus (Section 3.1). Subsequently, Section 3.2 describes a prototype of a *reactive model* for aggregate computing, introducing the objectives and giving a preliminary analysis of how the proposed model is expected to fulfill them. Finally, Section 3.3 presents the major design choices that were taken prior to implementation.

3.1 Analysis of the state of the art of aggregate computing

In order to have a comprehensive view on the subject, this section provides a brief analysis of the current state of aggregate computing, first by describing the *proactive model* and identifying its limitations, then by introducing *ScaFi* [3].

3.1.1 Proactive model

At the current stage, field calculus and aggregate computing are based on a model where each device of the network *repeatedly* executes its computation in rounds of *sense-eval-broadcast*. In particular, referring to the model discussed in [11], each sense-eval-broadcast round of a device is alternated with some *sleeping time* during which it collects information from neighboring devices. This way of managing computation can be thought of as a *proactive model*, since its the device that decides when computation should occur based on its internal scheduler.

The proactive model has some shortcomings. On the one hand, there is no way to granularly control the *timing* and the *dynamics* of computation rounds using the main operators of field calculus alone. In fact, there is often the need to

Listing 3.1: The constructs of the core API of ScaFi.

```scala
trait Constructs {
  def nbr[A](expr: => A): A
  def rep[A](init: =>A)(fun: (A) => A): A
  def foldhood[A](init: => A)(aggr: (A, A) => A)(expr: => A): A
  def aggregate[A](f: => A): A
  def align[K,V](key: K)(comp: K => V): V

  def mid(): ID
  def sense[A](name: CNAME): A
  def nbrvar[A](name: CNAME): A
}
```

perform computation upon the occurrence of particular events, for example when a sensor changes its value, or when a message is received from a neighbor. On the other hand, computations are carried out regardless of the existence of significant changes in the environment or in the knowledge of the neighborhood. In turn, this implies that the *broadcast* step is also carried out even if there is no change in the generated export, resulting in *wasteful message exchange*.

The shortcomings of the proactive model call for a more *reactive approach*, where taking actions only upon significant changes is a pivotal concern and should be taken into account as first class in the supporting model. A prototype for this is presented in Section 3.2.

3.1.2 ScaFi

This section presents a brief introduction to **ScaFi**, a Scala-based library and framework for aggregate programming [8]. In particular, the *API* and the *core* of ScaFi will be analyzed in order to facilitate both the design and implementation stages, since they can be used as references to guide the whole process.

ScaFi's API is heavily inspired by the field calculus' language and consists of a trait defining the main constructs that can be used to describe aggregate computations (Listing 3.1) Some notes to keep in mind about ScaFi's API are:

- an expression written using the API is evaluated by each device once per computation round;
- fields are represented as *atomic values* (i.e., they have no particular wrapper around them) and they indicate the value of the field at the device performing that computation;
- `sense` evaluates the sensor with the given `name`, effectively producing a field of the requested sensor value;

Listing 3.2: Implementation of a gradient using ScaFi

```scala
def gradient(src: Boolean): Double =
  rep(Double.PositiveInfinity) { distance =>
    mux(src) {
      0.0
    } {
      minHoodPlus(nbr(distance) + nbrRange)
    }
  }
```

- `nbrvar` is a mechanism to perform a query against a neighboring sensor with a given `name`;
- the concept of neighboring field from field calculus is not "reified", meaning that there is no actual data structure representing it; spatial computation (i.e., the `nbr` and `nbrvar` constructs) is only available inside a special scope, given by the `foldhood` construct;
- `foldhood` acts in such a way that the `expr` parameter is evaluated for each aligned neighbor (internally constructing the neighboring field) and the final output is obtained by *folding* all neighboring values using `init` and `aggr`;
- the export for each iteration is constructed by the *engine* of ScaFi, by applying side-effects to an internal data structure as these constructs get invoked, therefore constructing the evaluation tree.

This API can be used to implement an idiomatic building block of aggregate computing, which is known as *gradient*. A gradient is a numerical field that expresses the minimum distance from any device to source devices (Figure 3.1). The implementation of a gradient using ScaFi is presented in Listing 3.2. Some notes about the implementation:

- `src` is an input field of source devices;
- the `mux` operator acts like an if statement where both branches get evaluated and end up in the export (unlike the `branch` construct, that only evaluates the side selected by the condition); this means that both source and non-source nodes will be aligned regardless of the chosen branch;
- `minHoodPlus` is an operator implemented in terms of `foldhood`, finding the minimum value for the given expression among neighbors;
- the `Plus` suffix of `minHood` indicates that the device itself is not considered during the calculation of the minimum, which for the gradient has the effect

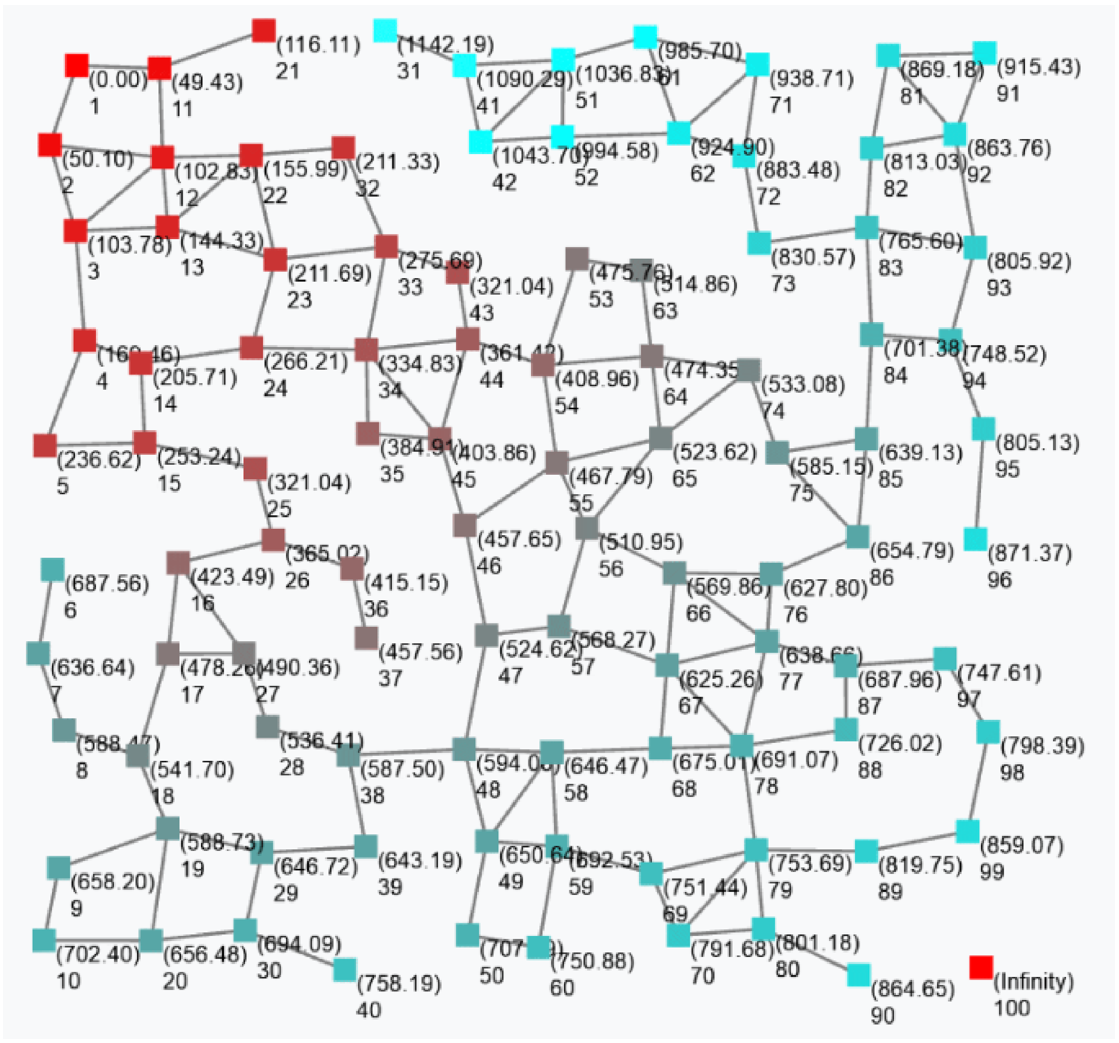


Figure 3.1: A graphical representation of a gradient after stabilization. Each device of the network is labelled with its distance from the source (in parenthesis) and its ID. The source device is the one with ID 1. Note that devices that are not connected to the source are considered to be at an infinite distance from it.

of preventing devices from getting stuck on low values after a source gets deactivated;

- `nbrRange` is a built-in neighboring sensor (implemented in terms of `nbrvar`) returning the estimated distance to the neighbor against which it is evaluated;
- source devices are at distance 0 from themselves, therefore the "then" branch of `mux` returns `0.0`;
- at each device the gradient is calculated by repeatedly minimizing, for every neighbor, the sum of its currently estimated distance from the source (`nbr(distance)`) and the distance between the device and the neighbor (`nbrRange`).

3.2 Reactive model proposal

As stated before, in order to overcome the limitations of the proactive model this thesis proposes an approach based on *reactivity*, leveraging the power of FRP (Section 2.4) to deal with the complexity of the approach. The sections below illustrate the objectives that are expected to be fulfilled by the final implementation and a high level description of the proposed reactive model.

3.2.1 Objectives

The high level goal of this thesis is to provide a model that is expressive enough to allow developers to declaratively describe *self-organizing* aggregate computations, while treating the dynamics and timing of relevant events as first class citizens. This vision can be in fact summarized with *functional reactive self-organization.*

The main objectives to be pursued in order to accomplish the goal are:

- **Compute only upon relevant changes**: computations should occur reactively only when something changes in the environment, in order to avoid wasteful resources usage;
- **Broadcast messages only upon relevant changes**: each device should avoid broadcasting an export that did not change since the last one, with the direct consequence that no further message exchange should be required if a computation reaches a stable configuration;
- **Avoid re-evaluation of unaffected sub-computations**: if a portion of the computation depends on data that did not change, it should not be re-evaluated.

3.2.2 Differences from the proactive model

The main difference between the model proposed by this thesis and the one proposed by field calculus is the *absence of computation rounds*. This is dictated by the fact that one of the objectives is to have computation run only upon relevant changes in the environment, namely:

- a new device enters the neighborhood;
- a device leaves the neighborhood;
- an export coming from a neighbor is received;
- the value of a sensor changes;
- the value of a neighboring sensor changes.

Previously, these sources of events were all handled in the *sense* stage of a computation round, and were all collected together in order to construct the *context* for the round itself. However, this was not done in a reactive fashion, since all these events were queued up while the device was sleeping and handled all at once at the start of the sense phase. This meant that, if no event was received between two rounds, the computation would still happen. The reactive model, instead, handles events as soon as they are received by the device (and only in that occasion), and only broadcasts the corresponding export if it is different than the previous, effectively fulfilling the "compute only upon relevant changes" and "broadcast messages only upon changes" objectives.

For what concerns the last objective (i.e., avoiding re-evaluating unaffected sub-computations), the optimized propagation of changes will be delegated to the correct use of the FRP engine.

3.3 Specification of the reactive model

This section analyzes the key elements of the prototypal design for the reactive model. Note that at this stage, the discussion is not tied to any particular technology, framework o programming language and should therefore be considered a *specification* rather than a mirror of the actual code. A deeper view on how this specification was translated into code can be found in Chapter 4. There is, however, a reference to Scala's data structures (e.g., `Maps`, `Options`) and syntax for generics, which should not be considered as an enforcement on the usage of the Scala language.

Figure 3.2 shows a holistic picture of the proposed design, formalized as a UML class diagram. The diagram does not show definitions for the types named

DeviceId, LocalSensorId and NeighborSensorId. This is a deliberate choice, since the specification does not depend on what those types really are. Their only constraint is that it must be possible to tell if any two values of those types are equal with each other.

The core of the specification starts from the definition of the Context, which is a way to express the environment where a device is immersed, and to provide handles for environmental changes to the computation. In fact, this is done by expressing values that are subject to changes throughout the lifecycle of the system as cells. In particular, sources of changes are:

- local sensors
- neighbor devices information, which is expressed as a time-varying map from device identifiers to the last registered NeighborState, namely:
  - the value for each neighboring sensor;
  - the last export that was received.

On top of that, the context is tied to the device identifier with which it is associated.

The Export that can be emitted by each device is represented as a tree where each child of a node is associated to a Slot that is unique among its siblings. Being modeled this way, each sub-tree of the export is identified by a *path* (i.e., a sequence of Slots) starting from the root (an *empty path*) and traversing the tree down to the node where it is rooted. An example of this is shown in Figure 3.3, which depicts the export of the following expression:

```
branch(
  sensor[Boolean](SomeSensor),
  mux(
    sensor[Boolean](AnotherSensor),
    constant(1),
    constant(2)
  ),
  constant(0)
)
```

in a situation where SomeSensor evaluates to true and AnotherSensor evaluates to false. Note that, for now, the semantics that lead to this particular shape of export are not relevant, as they will be discussed in detail in Section 3.3.1. It is sufficient to know that each syntactic element of the expression above (with the exception of constant(0) which is excluded due to branching) corresponds to

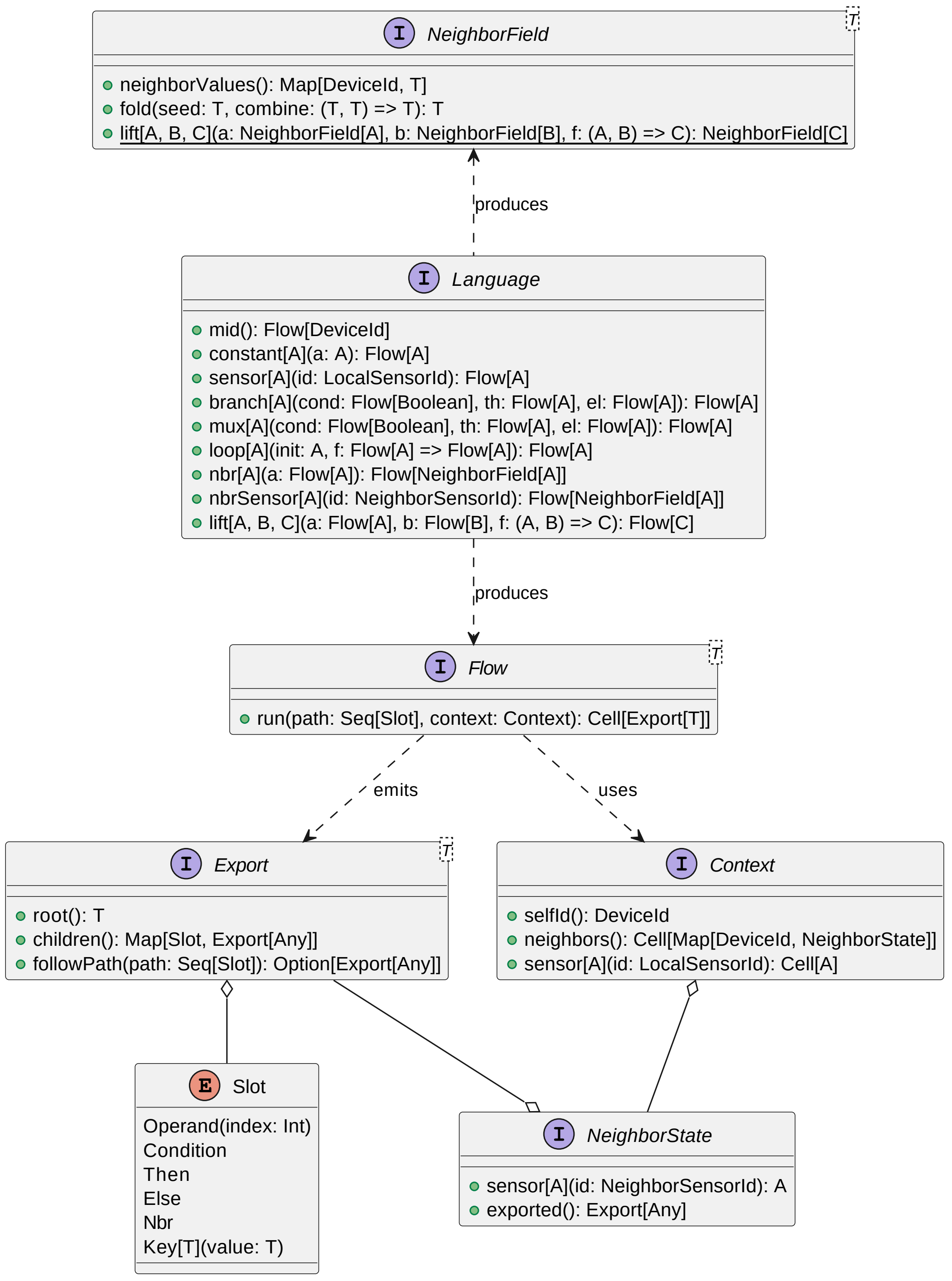

NeighborField
T
neighborValues(): Map[DeviceId, T]
fold(seed: T, combine: (T, T) => T): T
lift[A, B, C](a: NeighborField[A], b: NeighborField[B], f: (A, B) => C): NeighborField[C]
produces
Language
mid(): Flow[DeviceId]
constant[A](a: A): Flow[A]
sensor[A](id: LocalSensorId): Flow[A]
branch[A](cond: Flow[Boolean], th: Flow[A], el: Flow[A]): Flow[A]
mux[A](cond: Flow[Boolean], th: Flow[A], el: Flow[A]): Flow[A]
loop[A](init: A, f: Flow[A] => Flow[A]): Flow[A]
nbr[A](a: Flow[A]): Flow[NeighborField[A]]
nbrSensor[A](id: NeighborSensorId): Flow[NeighborField[A]]
lift[A, B, C](a: Flow[A], b: Flow[B], f: (A, B) => C): Flow[C]
produces
Flow
T
run(path: Seq[Slot], context: Context): Cell[Export[T]]
emits
uses
Export
T
root(): T
children(): Map[Slot, Export[Any]]
followPath(path: Seq[Slot]): Option[Export[Any]]
Context
selfId(): DeviceId
neighbors(): Cell[Map[DeviceId, NeighborState]]
sensor[A](id: LocalSensorId): Cell[A]
Slot
Operand(index: Int)
Condition
Then
Else
Nbr
Key[T](value: T)
NeighborState
sensor[A](id: NeighborSensorId): A
exported(): Export[Any]

Figure 3.2: A class diagram of the specification of the reactive model.

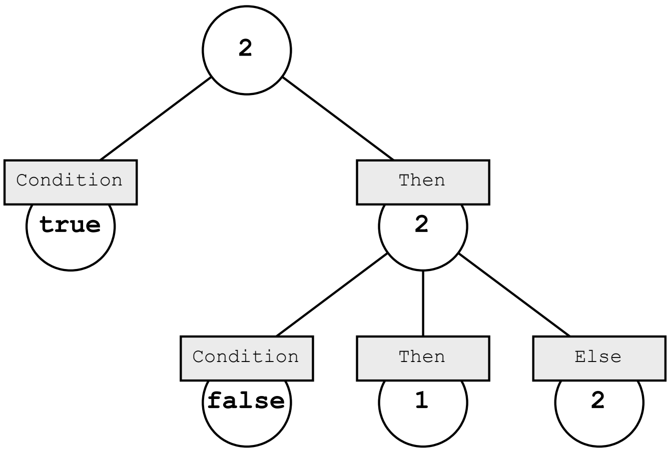


Figure 3.3: An example of an export. Circles represent nodes, while rectangles represent slots.

some path inside the tree, as follows:

`[]`	⇔	`branch(...)`
`[Condition]`	⇔	`sensor[Boolean](SomeSensor)`
`[Then]`	⇔	`mux(...)`
`[Then / Condition]`	⇔	`sensor[Boolean](AnotherSensor)`
`[Then / Then]`	⇔	`constant(1)`
`[Then / Else]`	⇔	`constant(2)`

This representation of an export is essential to perform the *alignment* process. In fact, two exports are aligned at a certain path if and only if that path exists in both exports.

All the constructs of the `Language` deal with `Flows`, a type which is designed to encapsulate aggregate (sub-)computations as a dependency graph (in a way that's closely related to the dependency graph of an FRP engine). A `Flow`, in fact, is essentially a function that takes a path and a `Context` and returns a cell of `Exports`, possibly depending on the exports of other `Flows` recursively. The path represents the point in the evaluation tree where that `Flow` is placed, while the `Context` places the computation inside some device. Notice that this design of a `Flow` has some consequences:

- since the context is passed to it after its construction and the output is a cell, a `Flow` effectively represents an entire aggregate computation, independent of space and time;

- since exports are returned as a cell, all (sub-)computations are automatically updated when the original sources of events are updated by the context;
- due to the fact that the path is also passed when requesting the cell of exports, the same instance of a `Flow` can be used multiple times in different points of the same computation, making it referentially transparent;
- since FRP is used under the hood, the requirement of functions passed to operators of the language being referentially transparent still exists;
- the main expression of an aggregate program should be evaluated from the empty path.

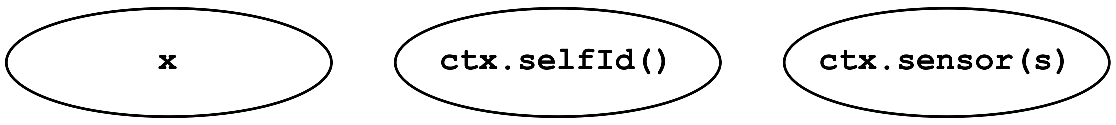
An important difference from the model used by ScaFi is the presence of a *reified* `NeighborField` type, that is used as the returned type of operators that deal with the neighborhood, i.e., `nbr` and `nbrSensor`. A `NeighborField` is essentially a wrapper around a map from neighbor identifiers to local values, which can be folded over in order to collapse all local values into a single one.

3.3.1 Constructs semantics

This section presents the semantics of the constructs belonging to the `Language` interface. Here, the term semantics is used to refer to the way the flow generated by each construct produces its exports from its inputs, depicted in a visual fashion. Note that these rules are to be interpreted as snapshots, and thus they should be re-applied on each new evaluation of the corresponding cell.

Some graphical notations are:

- expressions may use variables to capture arguments that are passed;
- a variable typed as a `Flow` may be used in a graph as the current export for that flow at the path where it is rooted;
- sub-exports may be represented as triangles when their content is not relevant;
- `r(_)` expresses the value contained in the root of an export;
- `N(_)` expresses the collection of all the given expressions for each aligned neighbor (graphically shown with arrows);
- `E(_)` denotes an atomic export whose root is the given argument;
- `ctx.xyz` corresponds to member `xyz` of the `Context` on which the flow is run;



(a) `constant(x)` (b) `mid()` (c) `sensor(s)`

Figure 3.4: Semantics of the atomic constructs.

- `nbr.xyz` refers to member `xyz` of the current neighbor being evaluated inside of `N(_)`.

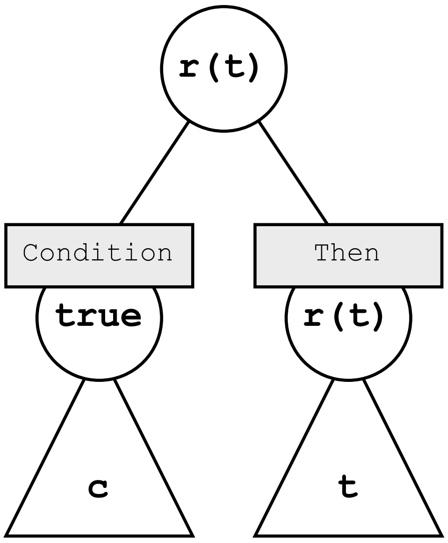
The simplest constructs of the language are those that are local and atomic (i.e., they work without knowledge of neighbors and they do not depend on other flows). These are described in Figure 3.4. Being atomic, their corresponding trees only contain a root node with the final value of the expression:

- `constant` defines a constant flow and always evaluates to the argument that has been passed;
- `mid` is a constant flow of device identifiers always evaluates to the `selfId` returned by the `Context`;
- `sensor` implements the sensing abilities of each device uses the cell returned by the `Context` for the given sensor identifier;

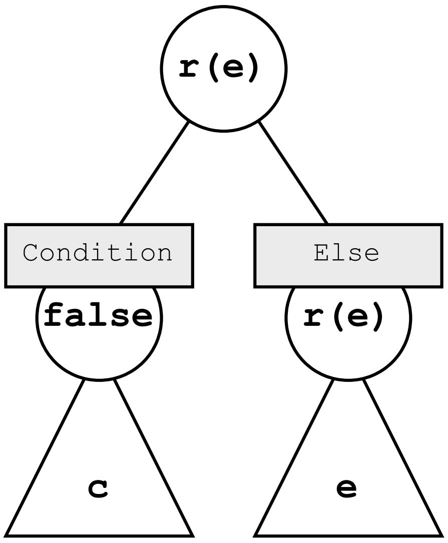
The `branch` construct (Figure 3.5) corresponds to the *if* construct of field calculus and works in such a way that, for a given expression `branch(c, t, e)`, only the export selected by `c` is included in the final output. In cases where the condition is true, `t` is included under the `Then` slot and its root is used as the global root, otherwise `e` is included under the `Else` slot and its root is used instead. In both cases, `c` is inserted under the `Condition` slot. With this semantics, any sub-computation under `t` or `e` will not align with any neighbor executing the opposite branch, due to the fact that their paths will differ at the `Then`/`Else` slot.

The `mux` operator (Figure 3.6) is an alternative to `branch` that introduces conditional behavior without resulting in partitions in the device network. That is, devices running sub-computations in `t` or `e` will always align with each other since both exports will be included in the final one, respectively under the `Then` and `Else` slots.

Neighboring constructs, i.e., `nbr` and `nbrSensor`, require knowing which neighbors are aligned when they get evaluated, as shown in Figure 3.7. This means that, if this constructs are used at a certain path `p`, the engine should filter out all neighbors whose known export does not contain `p`. Furthermore, only the sub-exports rooted at `p` of each neighbor need to be considered. The `nbr(a)` constructs works

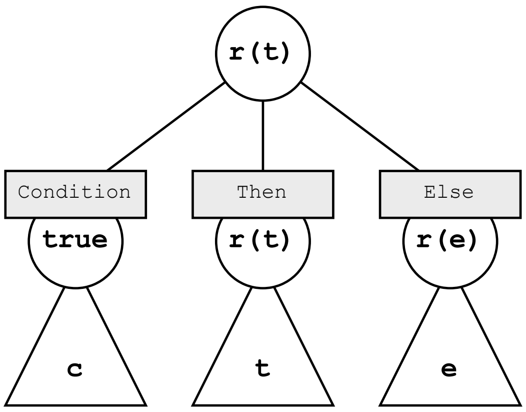


(a) `true` condition.

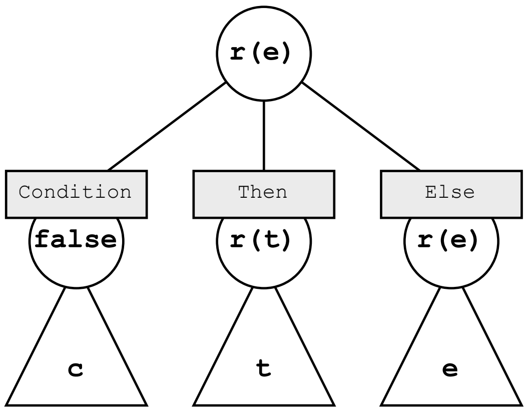


(b) `false` condition.

Figure 3.5: Semantics of `branch(c, t, e)`.



(a) `true` condition.



(b) `false` condition.

Figure 3.6: Semantics of `mux(c, t, e)`.

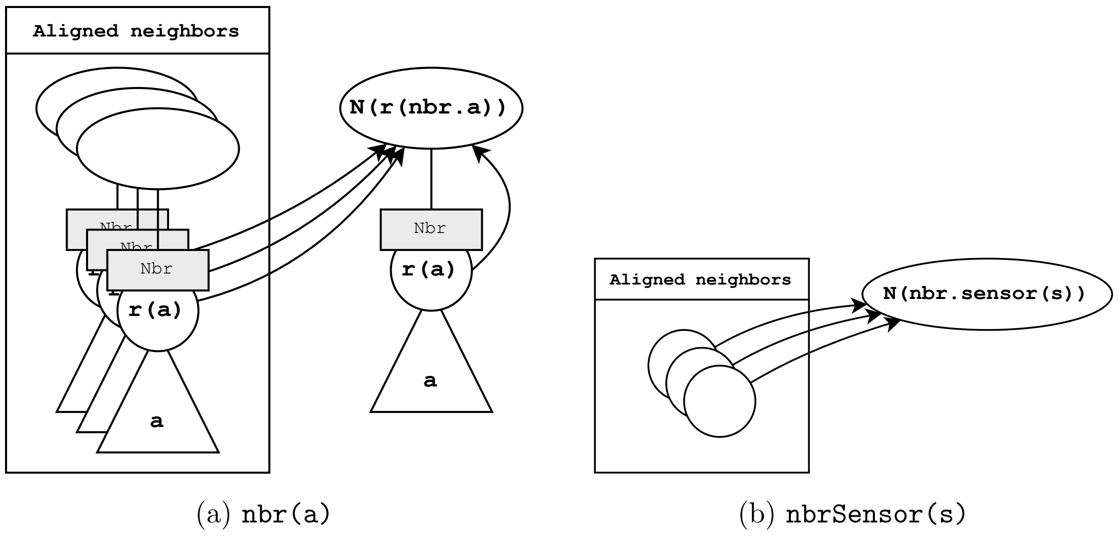


Figure 3.7: Semantics of neighboring constructs.

by collecting all neighboring values of `a`, which can be found at `p / Nbr` for each neighbor. Since every device is a neighbor of itself, this process should also take care of replacing the value produced by the previous evaluation with the new value coming from `a`. This is shown in Figure 3.7a by the isolated arrow going from `r(a)` in the current device to the root of the export. For what concerns `nbrSensor`, the engine just uses the current `NeighborState` of each aligned neighbor to evaluate the sensor with the specified identifier.

The language also defines a `lift` construct. *Lifting* is a well-known concept of functional programming that allows a function working with atomic values to be applied to wrapped versions of those values, for some wrapper type. In this case, the wrapper type is the `Flow` type, and lifting a function to the `Flow` world means producing another `Flow` obtained by applying that function to the roots of those flows, and having the original exports as children of the resulting one. This is shown in Figure 3.8, which depicts the case where `f` is a binary function (the generalization to n-ary functions is in fact trivial).

As a side note, the specification also defines a `lift` operation on instances of `NeighborFields`. It has the effect of evaluating the given function on pairs of values associated with the same device and returning the outputs as another `NeighborField`.

The final construct is `loop`. This is designed to be an adaptation of `rep` as defined by field calculus to the reactive model. This change was necessary due to the fact that `rep` is based on the underlying execution model structured in computation rounds, but that's not the case in the reactive model. Instead of using proactive and repeated application of a function, `loop` manages field

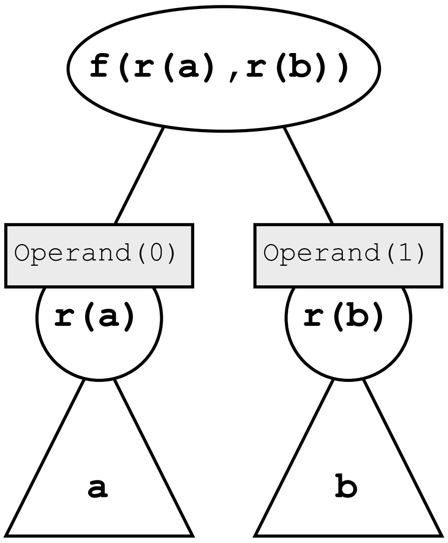
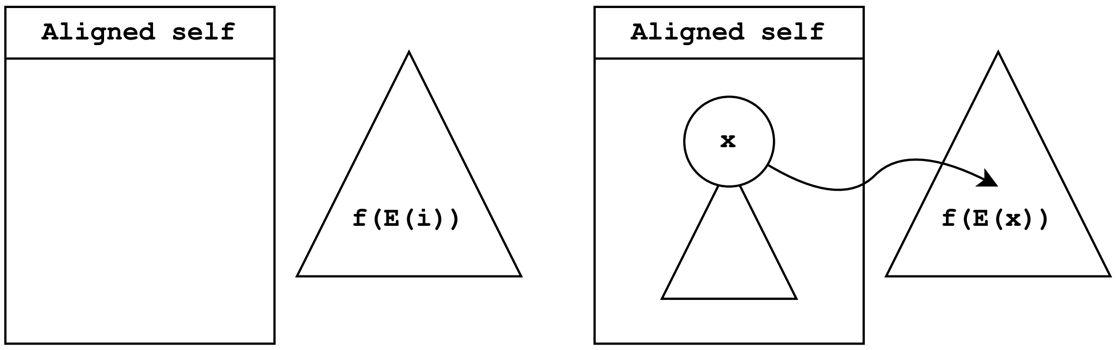


Figure 3.8: Semantics of `lift(a, b, f)`.

evolution over time as a reaction to changes in the previous state. Other than the initial state, `loop` accepts a function transforming a `Flow` into another `Flow` of the same type. These `Flows` shall be interpreted as the input always being "one step behind" the output. The specification states that the "off-by-one" `Flow` should be constructed by aligning with the previous state of the same device, which can be read from the context leveraging the fact that the every device is a neighbor of itself (Figure 3.9b). In cases where alignment is not possible (i.e., on the first evaluation or after a switch of branch), the first argument of loop should be used to construct a new export (Figure 3.9a). The implementation should take care of the following caveats arising from the intrinsic self-dependency of `loop`:

- *infinite recursion*: since computations are triggered by themselves changing in the past, `loop` should not cause *stack overflows* and therefore should use some sort of *rate-limiting* strategy (e.g., *throttling*);
- *indefinite export growth*: a flow defined in terms of itself might cause the export tree to grow on each update, since exports are are repeatedly wrapped in other exports; the implementation should take care of passing an *atomic* flow as an input to the subsequent computation, in order to avoid indefinite growth.



(a) Not aligned with previous state (b) Aligned with previous state

Figure 3.9: Semantics of `loop(i, f)`.

Chapter 4

Implementation

This chapter discusses the implementation of a proof of concept concretizing the specification introduced in Chapter 3.

The language of choice was Scala, primarily for the following reasons:

- its advanced features (e.g., mixins, family polymorphism, higher kinded types, etc) and flexible syntax make it a really powerful language to express DSLs (Domain Specific Languages) with a few lines of code;
- Scala 3's new features, in particular *context functions* and *extension methods*, allow for an even more concise syntax;
- ScaFi could be used as a guide throughout the development process, so it was possible to mimic its internal structure using similar constructs offered by the language.

4.1 Architecture

The project is structured into layers following the diagram shown in Figure 4.1. Their responsibilities can be summarized as follows:

- the **FRP** layer is the one exposing the FRP engine (as described in Section 2.4, using the Sodium library), also adding some extensions on top of that;
- the **Core** layer is the beating heart of the library, modeling and implementing the specification of Section 3.3;
- the **Simulation** layer provides a basic simulator capable of running aggregate programs for a network of devices.

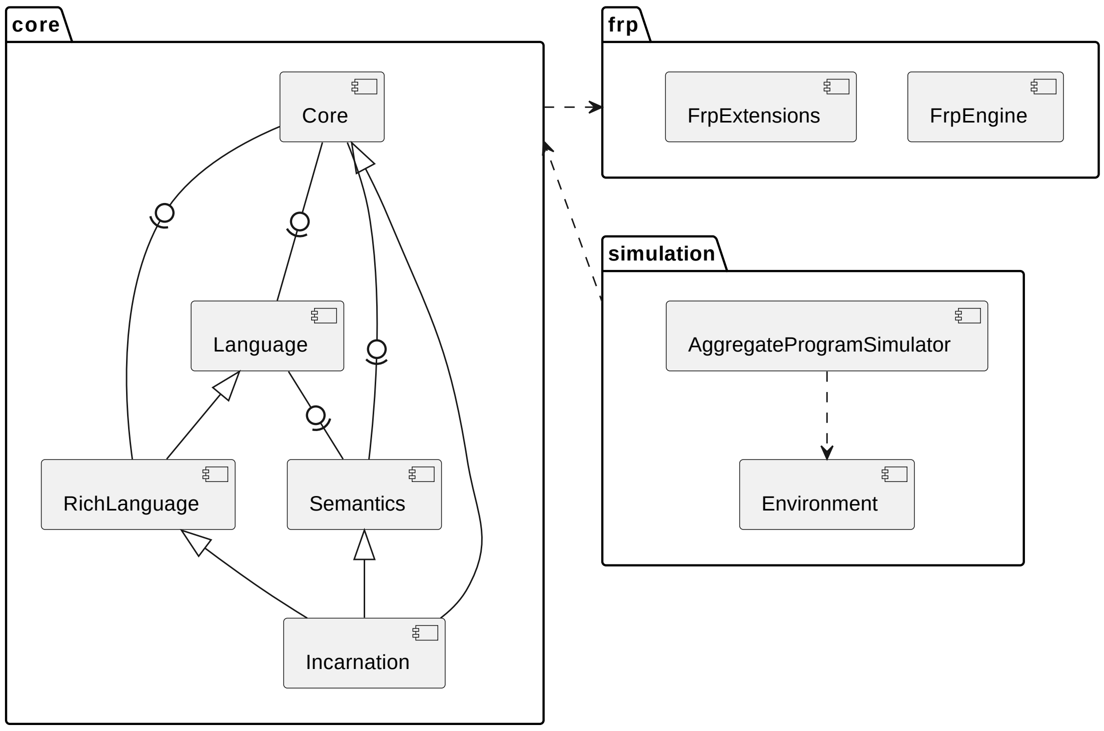


Figure 4.1: High level architecture, corresponding in part to the package structure.

Each layer, along with its sub-components, is described in greater detail in the sections below.

4.2 FRP layer

The FRP layer deals with cells and streams in order to provide expressive building blocks to the *core* layer. As stated before, this layer was implemented by leveraging the Sodium library and adding some extension methods and utilities to be used while defining the behavior of the aggregate constructs.

Some of these extensions include:

- *calming*: a way to create a cell whose listeners are only notified of an update if the previous value is different from the new one, for performance reasons;
- *buffering*: refers to a mechanism by which events of a stream are collected for a certain amount of time before being emitted all at once, using some combination of the events that were buffered;
- *throttling*: a particular way of buffering which combines element of the buffer by returning the last one that was received.

4.3 Core layer

The core layer follows an organization that is highly inspired to the core of ScaFi. In fact, it models the entire specification as being broken down into *components*, each providing a narrow set of features that can be combined together. A way to model component-based structures using Scala is through *mixins* and *self-types*, which were used extensively during the development of the core layer. These features, along with Scala's *abstract type members*, enable the use of *family polymorphism* [4] to model complex domains. In particular, the structure of the model presented in the specification was re-adapted in order to fit in the idiomatic practices of Scala and to be implemented via family polymorphism. This guarantees a greater flexibility when introducing changes to the specification and allows extensions of the language to be introduced with little or no work at all.

4.3.1 Core

The root of the domain is given by the `Core` trait (Listing 4.1). It defines the main concepts of an aggregate computation, namely:

Listing 4.1: The `Core` trait, the root of the family polymorphism.

```scala
trait Core:
  type context
  type Export[+_]
  type Path

  trait Flow[A]:
    def run(path: Path)(using Context): Cell[Export[A]]
```

- an abstract `Context` type, since the specific way to interact with the environment is unknown at this point;
- an abstract `Export[+_]` type, representing the output of a device with a `covariant` root type;
- an abstract `Path` type, denoting paths that can be looked up inside an `Export`;
- the `Flow[A]` trait, defined in terms of the abstract types above in a way that is almost identical to the one defined in the specification, with the only difference being that the `Context` is passed implicitly.

This is the minimum required information that is used to perform an aggregate computation.

4.3.2 Language

Below `Core`, the `Language` trait introduces the constructs when mixed in with a `Core` object, from which the basic abstract types of aggregate computing are imported. This is done via self-types, to avoid creating a strong inheritance hierarchy. The `Language` trait is shown in Listing 4.2 It introduces new abstract types that are specific to the constructs, in particular:

- `DeviceId` denotes an identifier for devices participating in the aggregate computation;
- `LocalSensorId` and `NeighborSensorId` identify sensors that can be queried locally or from any neighbor, respectively;
- `NeighborField[+_]` represents a mapping from DeviceIds to neighboring values of the given type.

A notable difference in how the language supports lifting operations (recall Section 3.3.1) is the fact that it is supported via a *type class* and *ad-hoc polymorphism.*

Listing 4.2: The Language trait, introducing the main API of aggregate computing.

```
trait Language:
  self: Core =>

  type DeviceId
  type NeighborField[+_]
  type LocalSensorId
  type NeighborSensorId

  val flowLiftable: Liftable[Flow]
  val neighborFieldLiftable: Liftable[NeighborField]
  given Liftable[Flow] = flowLiftable
  given Liftable[NeighborField] = neighborFieldLiftable

  def mid: Flow[DeviceId]
  def constant[A](a: A): Flow[A]
  def sensor[A](id: LocalSensorId): Flow[A]
  def branch[A](cond: Flow[Boolean])(th: Flow[A])(el: Flow[A]): Flow[A]
  def mux[A](cond: Flow[Boolean])(th: Flow[A])(el: Flow[A]): Flow[A]
  def loop[A](init: A)(f: Flow[A] => Flow[A]): Flow[A]
  def nbr[A](a: Flow[A]): Flow[NeighborField[A]]
  def nbrSensor[A](id: NeighborSensorId): Flow[NeighborField[A]]

  extension[A] (field: NeighborField[A])
    def foldLeft[T](seed: T)(combine: (T, A) => T): T
    def withNeighbor(id: DeviceId, value: A): NeighborField[A]
    def withoutNeighbor(id: DeviceId): NeighborField[A]

trait Liftable[F[_]]:
  extension[A] (a: F[A]) def map[B](f: A => B): F[B]
  def lift[A, B, C](a: F[A], b: F[B])(f: (A, B) => C): F[C]
  def lift[A, B, C, D](a: F[A], b: F[B], c: F[C])(f: (A, B, C) => D): F[D]
```

This was done by defining a `Liftable[F[_]]` trait defining that a wrapper type `F[_]`, generic in the type it encloses, supports lifting operations. As stated before, lifting is a way for a function operating on atomic values to be applied to wrapped versions of those values, and this can be generalized to any function regardless of the number of parameters. `Liftable[F[_]]` supports lifting for functions that take up to three parameters. However, lifting a unary function is more often `map`, and it is idiomatically more familiar when called as a method of the wrapper to be mapped, rather than as a top-level function. This is why `map` is defined as an extension method while the binary and ternary versions are not.

The next lines of the `Language` trait declare the constructs of the language, maintaining the same API as defined by the specification. In addition, since `NeighborField` is declared as an abstract type without any bounds, the language includes some of the operations that are needed, in particular by the `RichLanguage` trait. In fact, `RichLanguage` introduces utilities that are built on top of the main API, which are primarily common folding operations on flows of neighbor fields, like `min` or `toSet`.

4.3.3 Semantics

The `Semantics` trait starts reifying the concepts that are defined by the `Core` and by the `Language`, giving an implementation of the constructs that corresponds to the one proposed in Section 3.3.1. A portion of this trait can be found in Listing 4.3, where some of the abstract types that were still unbound up to this point get reified, in particular:

- `NeighborField[+A]` is reified as a `Map` from `DeviceId`s to `A`s;
- `Export[+A]` and `Path` are reified respectively as an `ExportTree[A]` and as a `Seq[Slot]`, for which the implementation is also included in the listing;
- `Context` gets bound to be a subclass of `BasicContext`, a new trait defined inside `Semantics`.

In addition, `Semantics` introduces a new abstract type `NeighborState` that is bounded to be a subclass of `BasicNeighborState`. This way of dealing with `Context` and `NeighborState`, i.e., by only giving upper bounds, allows for greater flexibility and type safety when implementing classes inheriting from `Semantics`, while still enforcing some constraints that are needed at this abstraction level.

The alignment process is implemented in the `alignWithNeighbors` method. In practice, this corresponds to the construction of a cell of maps that hold a certain value for all neighbors aligned at a given path. Each value is extracted using a function that takes the sub-export at which the alignment happened with that neighbor and the current `NeighborState`.

Listing 4.3: A portion of the `Semantics` trait reifying some of the domain's abstract types.

```scala
trait Semantics:
  self: Core with Language =>

  type NeighborState <: BasicNeighborState
  override type Context <: BasicContext
  override type NeighborField[+A] = Map[DeviceId, A]
  override type Export[+A] = ExportTree[A]
  override type Path = Seq[Slot]

  trait BasicNeighborState:
    def sensor[A](id: NeighborSensorId): A
    def exported: Export[Any]

  trait BasicContext:
    def selfId: DeviceId
    def sensor[A](id: LocalSensorId): Cell[A]
    def neighbors: Cell[Map[DeviceId, NeighborState]]

  ...

case class ExportTree[+A](root: A, children: Map[Slot, ExportTree[Any]]):
  def followPath(path: Seq[Slot]): Option[ExportTree[Any]] = path match
    case h :: t => children.get(h).flatMap(_.followPath(t))
    case _ => Some(this)

enum Slot:
  case Operand(index: Int)
  case Nbr
  case Condition
  case Then
  case Else
  case Key[T](value: T)
```

Listing 4.4: Implementation of the `nbr` construct.

```
object Flows:
  def of[A](f: Context ?=> Path => Cell[Export[A]]): Flow[A] = new Flow[A]:
    override def run(path: Path)(using Context): Cell[Export[A]] = f(path).calm
  ...

override def nbr[A](a: Flow[A]): Flow[NeighborField[A]] =
  Flows.of { path =>
    val alignmentPath = path :+ Nbr
    val neighboringValues = alignWithNeighbors(
      alignmentPath,
      (e, _) => e.root.asInstanceOf[A]
    )
    lift(a.run(alignmentPath), neighboringValues){ (exp, n) =>
      val neighborField = n + (ctx.selfId -> exp.root)
      ExportTree(neighborField, Nbr -> exp)
    }
  }
```

As an example of the usage of `alignWithNeighbors`, the implementation of the `nbr` construct is presented in Listing 4.4, which gathers the neighboring values located at `path + Nbr`, where `path` is the location where `nbr` is being evaluated. Subsequently, it replaces the old value computed at the previous evaluation with the new one exported by `a` before wrapping everything in a new export, following the semantics of `nbr`. Notice the adoption of a `Flows` factory to create the final result of `nbr`, in particular through the `of` method. This factory encapsulates the logic that constructs `Flow` instances, and it is used extensively while implementing all the constructs of the language to guarantee a consistent creation of flows. Also notice the use of `calm` in `Flows.of`, effectively avoiding the emission of the same export twice in a row. This is a key implementation choice that has a strong effect on the overall behavior and performance of the library, especially in relation to the objectives outlined in Section 3.2.1. The consequences of this choice will be examined deeply in Chapter 5.

4.3.4 Incarnation

An `Incarnation` represents the final concretion in the core layer, defining a unified composition of all the traits that have been described up until this point. Conceptually, it is a way to abstract away the details of a particular platform that is able to host an aggregate computation. Practically, this is done by providing a `factory` that creates instances of the `Context` given some device identifier. Its code is shown in Listing 4.5.

Note that, even at this level, some of the abstract types defined by `Language` are not bound to any type yet, in particular `DeviceId`, `LocalSensorId` and

Listing 4.5: The `Incarnation` trait.

```scala
trait Incarnation extends Core, RichLanguage, Semantics:
  def context(selfId: DeviceId): Context
```

`NeighborSensorId`. The nature of those types is in fact platform specific, thus it is up to the specific implementation of `Incarnation` to decide what those should be concretely.

4.4 Simulation layer

This layer introduces a way to test the behavior of an aggregate program by running it in a simulated environment. A simulation can in fact be carried out by creating *in-memory* versions of the `Incarnation` and the `Context`, using synthesized data to replace sensors.

4.4.1 Simulation incarnation

The incarnation implemented for the testing purposes of this thesis is based on a certain network topology defined by an `Environment` object. Simply put, an environment determines the number, the position and the neighboring relation of a device network in 2D space, assuming device identifiers are progressive integers.

In order to create a concrete `Incarnation` to be used in the simulator, several mixins were introduced to favor modularity and composition. These mixins provide vertical slices of independent features, each of which can be plugged in to build a complex domain out of simple components. In particular:

- the `IncarnationWithEnvironment` mixin provides the concept of an incarnation being situated in an `Environment`, while fixing the device identifiers to be integers;
- the `TestLocalSensors` mixin specifies that local sensors exist that can detect if a device is a *source* or an *obstacle*;
- the `TestNeighborSensors` mixin specifies the existence of a neighboring sensor that estimates distances.

These traits are all mixed in to compose the `SimulationIncarnation` and presented in Listing 4.6. The most relevant part of this incarnation in order to understand the simulator is the definition of the `SimulationContext`, in particular the strategy by which the context exposes the current neighbors' states

Listing 4.6: The `SimulationIncarnation` class.

```
class SimulationIncarnation(environment: Environment,
                            sources: Cell[Set[Int]] = new Cell(Set.empty),
                            obstacles: Cell[Set[Int]] = new Cell(Set.empty))
  extends Incarnation
    with IncarnationWithEnvironment(environment)
    with TestLocalSensors
    with TestNeighborSensors:
  ...
  class SimulationContext(val selfId: DeviceId) extends BasicContext:
    private val neighborsSink =
      IncrementalCellSink[Map[DeviceId, NeighborState]](
        Map.empty,
        calm = true
      )

    def receiveExport(neighborId: DeviceId, exported: Export[Any]): Unit =
      val newState = SimulationNeighborState(selfId, neighborId, exported)
      neighborsSink.update(_ + (neighborId -> newState))

    override def neighbors: Cell[Map[DeviceId, NeighborState]] =
      neighborsSink.Cell
  ...
```

to the core. Internally, the context uses a wrapper of Sodium's `CellSink` (i.e., `IncrementalCellSink`) that is able to perform incremental updates by applying a function to the current state. This is used to apply incremental updates to the state of neighbors as exports are received during program execution. By calling `receiveExport` when a new export is received from one of the neighbors of the device, the context can be informed of a new neighbor state that can be registered in the sink, triggering all computations that depend on that state.

4.4.2 Simulator

The final step towards simulating an aggregate computation is the simulator itself. The phases of a simulation can be summarized as follows:

1. **spawn contexts**: for each device participating in the computation, create a `Context` instance via the incarnation;
2. **run the flow**: run the `Flow` that is being simulated on all the context instances, producing a cell of `Exports` for each device;
3. **listen to export updates**: upon changes of any cell that was produced, broadcast that export to the contexts of neighboring devices.

Listing 4.7 shows the implementation of the `Simulator` class. Its constructor takes as inputs an instance of `SimulationIncarnation`, that will be used to

Listing 4.7: The `Simulator` class.

```scala
class Simulator(val incarnation: SimulationIncarnation,
                executor: ExecutorService = Executors.newSingleThreadExecutor):

  import incarnation._

  def run[A](flow: Flow[A]): Unit =
    val contexts = (0 until incarnation.environment.nDevices).map(context)
    Transaction.runVoid(() => {
      val exports = contexts.map { ctx =>
        (ctx.selfId, flow.run(Seq.empty)(using ctx))
      }
      exports.foreach((id, exp) => exp.listen(e => {
        println(s"Device $id exported:\n$e")
        incarnation.environment.neighbors(id).foreach { n =>
          executor.execute(() => contexts(n).receiveExport(id, e))
        }
      }))
    })
```

spawn context and to access the types defined by the model, and an optional `ExecutorService`, that is instead used to schedule notifications of export emissions in the background. This is useful for performance and concurrency reasons, but most importantly it is required since Sodium does not allow listeners to call `send()` on `CellSinks` (which is called internally by the `receiveExport` method). By scheduling these tasks on another control flow, this does not pose an issue.

Notice that steps 2 and 3 are wrapped inside a transaction. Other than guaranteeing that no "first events" are missed, wrapping both steps ensures that all flows start at the same logical instant, in a consistent fashion.

Chapter 5

Evaluation

This chapter discusses the strategies that were adopted in order to assess the fulfillment of the objectives defined in Section 3.2.1 and how they were satisfied by the implemented solution. Primarily, the chosen techniques were:

- **unit tests**, that verify the functional aspects of the project and define a formalization of the expectations about the semantics;
- **empirical tests**, which consist of *sample programs* run via simulation, in order to assess qualitative properties of the prototype.

5.1 Unit testing

In order to certify the correctness of the code written during development, the project was equipped with a suite of *unit tests* of the implementation of the semantics. Not only did this allow refactoring with greater confidence, but it also constituted a formalization of the expectations about the API of the language that is *verifiable* and *reproducible*.

In practice, unit tests were written using the *ScalaTest* [1] framework, one of the de facto standards for automated testing in Scala. Out of the many testing styles that it offers, the chosen one was the *FlatSpec*, due to its simple structure that promotes a behavior-driven approach to writing tests.

The most relevant tests that are included in the project are written for the `Semantics` trait. The starting point of the process was to implement a mocked version of the incarnation and the related context, in order to both have access to the constructs of the language and to be able to manually trigger state changes for the flows to react to. Being based on the same ideas as the `SimulationIncarnation`,

[1] `https://www.scalatest.org/`

the source code for the `MockIncarnation` is omitted. As an example of the principles that guided this process, Listing 5.1 shows a portion of the `SemanticsTests` class that includes tests for the `lift` construct.

Listing 5.1: A portion of the `SemanticsTests` class including tests for the `lift` construct.

```
class SemanticsTests extends AnyFlatSpec with should.Matchers with
    BeforeAndAfterEach:
  private val SELF_ID = 1
  private val PATH = Seq.empty
  ...

  object SemanticsTestsIncarnation extends MockIncarnation:
    ...

  import SemanticsTestsIncarnation._
  import SemanticsTestsIncarnation.given

  private given ctx: Context = context(SELF_ID)

  override def beforeEach(): Unit = ctx.reset()

  "lift" should "combine two flows by nesting them" in {
    val left = "LEFT"
    val right = "RIGHT"
    val flow = lift(constant(left), constant(right))(_ + _)
    flow.run(PATH).sample() should be (ExportTree(
      left + right,
      Operand(0) -> ExportTree(left),
      Operand(1) -> ExportTree(right),
    ))
  }

  it should "react to changes in either its inputs" in {
    val stringOp = new CellSink("A")
    val intOp = new CellSink(2)
    val flow = lift(Flows.fromCell(stringOp), Flows.fromCell(intOp))(_ * _)
    val exports = flow.run(PATH)
    exports.sample().root should be ("AA")
    stringOp.send("B")
    exports.sample().root should be ("BB")
    intOp.send(3)
    exports.sample().root should be ("BBB")
  }
```

Typically, unit tests are implemented by constructing a flow that includes the primitive that is being tested, running it with an empty path using the given context implicitly, and then sampling the exports, possibly after notifying some state change. The results of sampling are then asserted against some condition to verify that exports behave as expected.

5.2 Sample programs

While unit tests ensure the correctness of the semantics from a purely functional standpoint, they are not enough evaluate the overall behavior of the library and the complete fulfillment of the objectives. To this purpose, a series of sample aggregate programs, for which the expected outcome is known in advance, was developed.

Listing 5.2 shows the implementation of a comprehensive sample that showcases almost every feature of the language, i.e., a *gradient with obstacles*.

Listing 5.2: A sample aggregate program computing a gradient with obstacles.

```
def runGradientSimulation(environment: Environment,
                          sources: Cell[Set[Int]],
                          obstacles: Cell[Set[Int]]): Unit =
  val incarnation = SimulationIncarnation(environment, sources, obstacles)
  val simulator = Simulator(incarnation)

  import simulator.incarnation._
  import simulator.incarnation.given

  def gradient(src: Flow[Boolean]): Flow[Double] =
    loop(Double.PositiveInfinity) { distance =>
      mux(src) {
        constant(0.0)
      } {
        liftTwice(nbrRange, nbr(distance))(_ + _).withoutSelf.min
      }
    }

  simulator.run {
    branch(obstacle) {
      constant(-1.0)
    } {
      gradient(source)
    }
  }

@main def gradientSample(): Unit =
  val sourcesSink = IncrementalCellSink(Set(0))
  val obstaclesSink = IncrementalCellSink(Set(2, 7, 12))
  val environment = Environment.manhattanGrid(5, 5)
  runGradientSimulation(environment, sourcesSink.cell, obstaclesSink.cell)
```

The environment used by this sample is a 5x5 grid, where each device is a neighbor of the nearest device in each horizontal and vertical direction (Figure 5.1). In addition, the device with ID 0 is a source node, while devices 2, 7, and 12 are obstacles.

The gradient with obstacles sample can be built starting from a simple *gradient*, thanks to the compositionality of aggregate computing. The implementation of the gradient itself is just an adaptation of the one that is already known in the literature and from field calculus to the new API. On top of that, obstacles are introduced by branching using the `obstacle` sensor provided by the incarnation,

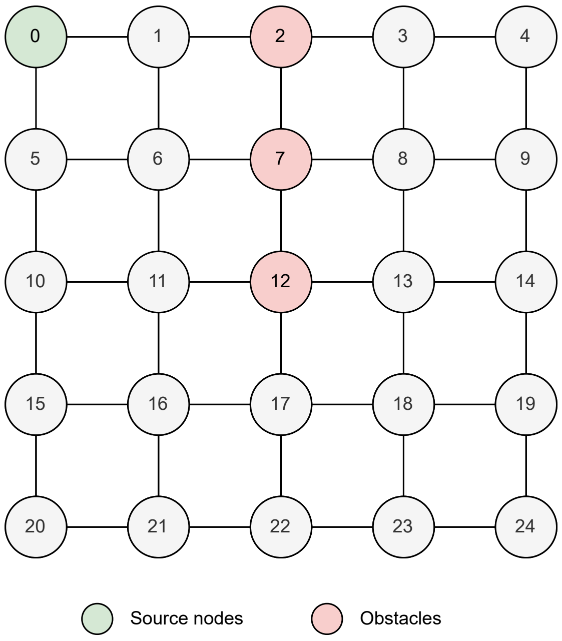


Figure 5.1: The reference environment for the gradient with obstacles.

using a default constant flow with value -1 where and when the sensor evaluates to true.

Notice that, when the sample is run, the application stops logging after a certain amount of time, indicating that no more exports are being generated. This demonstrates that the objective of "broadcasting messages only upon relevant changes" (recall Section 3.2.1) was fulfilled. In addition, by looking at the last export of each device, the global field that is produced is shown in Figure 5.2, which is the expected result.

To prove that also the "compute only upon relevant changes" objective was satisfied, a small addition to the sample was introduced. In fact, by placing these lines at the end of the `gradientSample()` method

```
Thread.sleep(5000)
sourcesSink.set(Set(4))
Thread.sleep(5000)
obstaclesSink.update(_ + 17)
```

the following behavior is exhibited:

- initially, the gradient is computed normally, and it self-stabilizes in less than 5 seconds, with the same output as the previous configuration;
- after 5 seconds from the start of the simulation, the aggregate of devices recognizes a change in the source field and re-adapts itself in a reactive way;

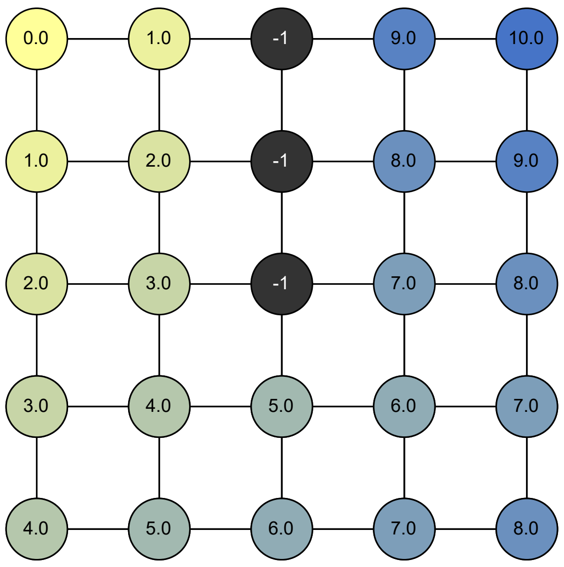


Figure 5.2: The output field of the gradient with obstacles sample, after stabilization.

- after 10 seconds, a new obstacle is detected by the network and the gradient adapts yet again.

Notice that, in this last scenario, updates do not reach portions of the network that are not influenced by the gradient change, implying that only computations that are strictly required are actually carried out.

For what concerns the last objective, i.e., “avoid re-evaluation of unaffected sub-computations”, another sample was created from scratch (Listing 5.3). This program starts a simulation on a single device with a flow that branches on the `source` sensor, where the “then” branch performs some intense computation on a constant flow. Subsequently, it changes the value of the source sensor back and forth in order to switch the selected branch alternatively. In ScaFi, the intense computation would be re-evaluated on each round where the value of `source` was true. To verify that this is not the case for this implementation, `someIntenseComputation` performs a side effect that prints to the console some text. This is, indeed, a violation of the principles of referential transparency of functions passed to the constructs of the language, but it serves the purpose of counting the number of calls that are made to that method. In fact, running this program results in “*Doing some intense computation...*” being printed only once, while the device broadcasts messages five times instead, demonstrating that sub-computations that are not affected by changes in the environment are not re-evaluated.

Listing 5.3: A sample aggregate program verifying that sub-computations do not get re-evaluated if their dependencies do not change.

```
@main def testReEvaluation(): Unit =
  val sourcesSink = IncrementalCellSink(Set.empty)
  val environment = Environment.singleNode
  val incarnation = SimulationIncarnation(environment, sources = sourcesSink.cell)
  val simulator = Simulator(incarnation)

  import simulator.incarnation._
  import simulator.incarnation.given

  def someIntenseComputation(input: String): String =
    println("Doing some intense computation...")
    input

  simulator.run {
    branch(source) {
      constant("I'm a source device").map(someIntenseComputation)
    } {
      constant("I'm not a source device")
    }
  }

  sourcesSink.update(_ + 0)
  sourcesSink.update(_ - 0)
  sourcesSink.update(_ + 0)
  sourcesSink.update(_ - 0)
```

Chapter 6

Conclusions

This thesis was meant to be an exploratory study of the applicability of FRP to the aggregate computing paradigm. All the objectives that were identified were achieved and the implemented library was verified to be compliant with the specifications, so the overall result can be considered a success.

Being a prototype, what has been implemented is far from being a complete and reliable solution for reactive aggregate programs. Nonetheless, this goes a long way in showing that a functional reactive approach to aggregate computing is certainly possible and that the benefits that it brings to the table are really valuable, therefore the following paragraphs include topics where future efforts might be directed in this regard.

Support for real world distributed platforms At the current stage, the library only supports being run on a simple simulator with very little features. Since the hope is that this prototype can one day evolve into a solution for large-scale distributed system, one of the future developments would certainly need to introduce support for real-world platforms in order to make deployments on real devices possible.

Core API improvements At a first glance, the new API introduces some noise to the overall structure, due to the fact that, differently from ScaFi, it operates on flows instead of local values. This in fact requires normal operators to be constantly lifted in order to be applicable to flows, introducing boilerplate code that makes programs less transparent. In the future, some efforts could be put into researching a better API to deal with lifting operators in a more scalable and user-friendly way.

Support for the Alchemist simulator The simulator that was developed to showcase the library at work is nowhere near an adequate solution to test systems with complex rules and behavior. Since *Alchemist* [7] already constitutes a solution to the necessity of a feature-rich simulator, a future version of the library might integrate with it and provide a simple way to test aggregate programs against complex simulations.

Improved timing control At the moment, the granularity with which the timing of computation can be configured allows no more than reactions to standard events coming from the context. It would be nice if the framework supported additional strategies for *scheduling* and *rate limiting* other than calming and throttling, maybe configurable per construct.

Bibliography

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4), aug 2013.

[2] S. Blackheath. *Functional Reactive Programming.* Manning, 2016.

[3] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala dsl and toolkit for aggregate programming. *SoftwareX*, 20:101248, 2022.

[4] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, pages 303–326, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[5] New in scala 3. `https://docs.scala-lang.org/scala3/new-in-scala3.html`. Accessed: 2023-02-22.

[6] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, apr 2014.

[7] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7, 01 2013.

[8] Scafi documentation. `https://scafi.github.io/`. Accessed: 2023-02-26.

[9] Scala 2.13 language specification. `https://www.scala-lang.org/files/archive/spec/2.13/`. Accessed: 2023-02-21.

[10] Scala 3 reference. `https://docs.scala-lang.org/scala3/reference/`. Accessed: 2023-02-21.

[11] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.*, 28(2), mar 2018.

[12] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109:100486, 2019.