FScaFi : A Core Calculus for Collective Adaptive Systems Programming

FScaFi: a Core Calculus for Collective Adaptive Systems Programming

Roberto Casadei[1][0000-0001-9149-949X], Mirko Viroli[1][0000-0003-2702-5702], Giorgio Audrito[2][0000-0002-2319-0375], and Ferruccio Damiani[2][0000-0001-8109-1706]

[1] Alma Mater Studiorum–Università di Bologna, Cesena, Italy
{roby.casadei, mirko.viroli}@unibo.it
[2] Università di Torino, Italy
{giorgio.audrito, ferruccio.damiani}@unito.it

Abstract. A recently proposed approach to the rigorous engineering of collective adaptive systems is the aggregate computing paradigm, which operationalises the idea of expressing collective adaptive behaviour by a global perspective as a functional composition of dynamic computational fields (i.e., structures mapping a collection of individual devices of a collective to computational values over time). In this paper, we present FScaFi, a core language that captures the essence of exploiting field computations in mainstream functional languages, and which is based on a semantic model for field computations leveraging the novel notion of "computation against a neighbour". Such a construct models expressions whose evaluation depends on the same evaluation that occurred on a neighbour, thus abstracting communication actions and, crucially, enabling deep and straightforward integration in the Scala programming language, by the ScaFi incarnation. We cover syntax and informal semantics of FScaFi, provide examples of collective adaptive behaviour development in ScaFi , and delineate future work.

1 Introduction

The Internet of Things (IoT), Cyber-Physical Systems (CPS), and related initiatives point out a trend in informatics where computation and interaction are increasingly pervasive and ubiquitous, and carried on by a potentially huge and dynamic set of heterogeneous devices deployed in physical space. To address the intrinsic complexity of these settings, a new viewpoint is emerging: a large-scale network of devices, situated in some environment (e.g., the urban area of a smart city), can be seen as a computational overlay of the physical world, to be programmed as a "collective" exhibiting robustness and resiliency by inherent adaptation processes. These kinds of systems are sometimes referred to as *Collective Adaptive Systems (CAS)* [2], to emphasise that computational activities are collective (i.e., they involve multiple coordinated individuals), and that a main expected advantage is inherent adaptivity of behaviours to unforeseen changes (e.g., as induced by changes/faults in the computational environment or interactions with humans or other systems).

Aggregate Computing [11] is an approach to CAS engineering that takes a global stance to design and programming and where coordinated adaptation is a key feature. Hence, it targets problems and application domains such as crowd engineering, complex situated coordination, robot/UAV swarms, smart ecosystems and the like [37]. Its key idea is to *program a large system as a whole*, that is, to directly consider an ensemble of devices as the target machine to be programmed, and provide under-the-hood, automatic global-to-local mapping: once the desired system-level behaviour is expressed by a global program, then individual computational entities of an aggregate are bound to play a derived, contextualised local behaviour of that program. Prominently, the distinguishing characteristic of Aggregate Computing as a "macro-approach" [10] lies in the ability to formally represent the adaptive behaviour of an ensemble *in a compositional and declarative way*, namely, by combination of functional coordination operators and high-level building blocks expressing the outcome of a collective task.

One fundamental enabling abstraction for specifying the dynamics of situated collectives is that of a *computational field* (or simply, field) [6, 8, 29]: a distributed data structure that maps devices to computational objects across time. Accordingly, *Aggregate Programming* is about describing (dynamic) field computations, namely, how input fields (data coming from sensors) turn into output fields (actions feeding actuators)—computations that can be conveniently expressed using the functional paradigm.

A modern implementation of the aggregate programming paradigm is ScaFi[3] (Scala Fields) [15]. It is a toolkit, tightly integrated with the Scala programming language, that comprises a Domain-Specific Language (DSL), a library, and platform tools for specifying and running (distributed) systems by leveraging computational fields. ScaFi provides a number of key advantages with respect to previous implementation attempts which were standalone DSLs (Protelis [35] and Proto [8]), such as: *(i) familiar programming environment*, by coherently supporting field constructs within the ecosystem as well as the syntactic and semantic model of a mainstream language like Scala; *(ii) lightweight type safety*, by leveraging Scala's powerful type system and type inference; and *(iii) seamless reuse of functionality*, by providing unrestricted access to both Scala features (e.g., lightweight components, implicits) and existing libraries on the JVM.

Technically, such a smooth integration with Scala has been achieved thanks to a semantic variation of previous formalisation attempts, which were based on the *field calculus* [6]: the notion of "neighbouring value" (a map from neighbours to data values), used in field calculus to locally express outcome of message reception from neighbours, is replaced with that of "computation against a neighbour", namely, by expressions whose evaluation depends on the same evaluation occurring on a neighbour. This change leads to a new computational model that we reify by a calculus called Featherweight ScaFi (FScaFi) and present in this paper.

[3] `https://scafi.github.io`

The content is structured as follows. Section 2 provides motivation for ScaFi and covers related work. Section 3 describes syntax and informal semantics of FScaFi. Section 4 provides examples, showing how FScaFi can be used to develop collective adaptive behaviour. Section 5 ends up the paper with a wrap-up and discussion of future work.

2 Background

Scenarios like the IoT, CPS, smart cities, and the like, foster a vision of rich computational ecosystems providing services by leveraging strict cooperation of large collectives of smart devices, which mostly operate in a contextual way. Engineering complex behaviour in these settings calls for approaches providing some abstraction through the notion of *ensemble*, neglecting as much as possible the more traditional view of focussing on the single device and the messages it exchanges with peers.

2.1 Aggregate Computing

Aggregate computing is the main theme of this paper. A recent survey of its historical development and state-of-the-art is provided in [37]. The essence of the approach is captured by the field calculus [6], a core language grounding semantics and formal analysis of field computations [12, 36].

Programming languages to work with computational fields have been introduced in the past, with Proto [8] as common ancestor (Lisp-based), and Protelis [35] as its Java-oriented, standalone DSL version. These approaches however have the drawback of not smoothly integrating field computations in the syntactic, semantic, and typing structures of a modern, conventional language—to fully remedy this problem, in this paper we shall present some key semantic changes to the field calculus.

To address this problem, a prominent, modern approach is to devise an "internal" or "embedded" DSL [42] that provides mechanisms to support the new features on top of an adequate host programming language. Of course, with embedded DSLs, both the syntax and the semantics are limited by the constraints exerted by the host language. However, the model can sometimes be slightly adjusted in order to favour the embedding, considering the common syntactic, typing, and semantic features of the candidate set of host languages.

When conceiving this DSL, we took into account the following requirements and desiderata for the host language: *pragmatism* (supporting easy reuse of existing programming mechanisms); *reliability* (intercepting errors early—cf., type checking); *expressivity* (offering an eloquent syntax); and *functional paradigm support* (all the significant features of functional programming must be cleanly available). All the above considerations led us towards the Scala programming language as the host. Then, to well design the key constructs and provide a framework for rigorous analysis of programs and properties, we came up with

FScaFi model of field computations. Its peculiarity is to handle standard values has been the local representative of a computational field, which provides a simplified setting for DSL embedding. To achieve this, we introduced a local notion of "computation against a neighbour", namely, a computation whose outcome depends on the most recent, local view of the result of computation in that neighbour (unlike in the standard field calculus, this allows smooth application of host typing mechanisms to any field expression)—as detailed in Section 3.

Such a model, and related tooling, is implemented in the ScaFi aggregate computing DSL and platform [15, 39]. ScaFi achieves the goal of providing an environment to streamline and support effective development of systems based on the Aggregate Computing paradigm, leveraging the solid basis provided by a mainstream programming language such as Scala and its ecosystem. In fact, Scala: runs on the JVM and thus enables straightforward interaction with the Java ecosystem; offers a powerful type system, with type inference, that helps to build type-safe libraries with minimal overhead; has a flexible syntax (convenient for creation of elegant APIs/DSLs). Moreover, Scala has great popularity in the distributed computing arena: it is the implementation language for several distributed computing toolkits, such as Apache Kafka[4] and the Akka actor framework[5]. Hence, our choice of Scala also fosters the construction of a platform-level support on top of ScaFi, in the form of a middleware for running distributed and situated systems [39, 16].

2.2 Related Work

Aggregate programming languages Prior aggregate programming languages are standalone (also called external) DSLs and include Proto [8], the Lisp-like progenitor, and its evolution Protelis [35]. Protelis is based on an untyped, standalone DSL able to interoperate with existing Java code. This approach has some limitations: aligning the syntax and semantics, as well as providing training and documenting for a *distinct* language w.r.t. the one used to develop the execution platform can be burdensome; extra development and maintenance effort is needed to adequately support editing tools (e.g., plugins are required for common IDE features like syntax highlighting and refactoring); activities that span both the DSL and the target language (e.g., static analysis and debugging) may be hard to implement; and finally, the ability to smoothly reuse the features and libraries of the target language can be limited. Though language tools greatly improved recently (cf. the Xtext language workbench [13] and its Xbase extension [22], to name a popular one), practically, with an external DSL it may be difficult to come up with a cohesive design of the resulting software system (cf. Generation gap pattern [41]), since parts written in the DSL need to bidirectionally refer and interact with other parts of the system [24].

[4] `https://kafka.apache.org`
[5] `https://akka.io`

Ensemble approaches In *Helena* [26] components can dynamically participate in multiple ensembles and adapt according to different *roles* whose behaviour is given by a process expression. *DEECo* [14] is another CAS model where components can only communicate by dynamically binding together through ensembles; DEECo ensemble is formed according to a *membership condition* and consists of one *coordinator* and multiple *members* interacting by implicit *knowledge exchange*; DEECo has a Java implementation called jDEECo which enables the definition of components and ensembles through Java annotations. The *GCM/ProActive* [7] framework supports the development of large-scale ensembles of adaptable autonomous devices through a hierarchical component model where components have a non-functional membrane and "collective interfaces", and a programming model based on active objects. *SCEL* [21] is a kernel language to specify the behaviour of autonomic components, the logic of ensemble formation, as well interaction through attribute-based communication (which enables implicit selection of a group of recipients). *Attribute-based communication* [1] is an approach to CAS coordination that leverages implicit multicasts towards recipients matched by predicates over attributes. The approach has been formalised by the *AbC calculus* [1] and implemented as an Erlang DSL in the so-called AErlang library [20]. Generally speaking, it is worth noting that the field calculus fits useful device abstractions (such as neighbourhood, message exchange, attribute-based filtering) into a purely functional approach, which can then smoothly interoperate with more traditional programming frameworks and languages. More specifically, attribute-based communication can be achieved in the field calculus (and hence in FScaFi) both at the receiver and the sender side, via construct `branch` (see Section 4), by which one can define subcomputations carried on by a subset of nodes—those that execute the same branch and hence remain actually "observable" by operator `nbr`. In a more programmatically expressible way, a notion of ensemble can be captured as a field computation on a dynamic domain of devices, denoted by the concept of an *aggregate process* [17].

Spatial computing and macro-programming An extensive survey on spatial computing can be found in [10]. Indeed, multiple classes of approaches address (at least in part) the problem of organising a collective of computational entities. These include *topological and geometrical languages* like *GPL* [18] (exploiting the botanical metaphor of "growing points") and *OSL* [32] (focussing on programming "computational surfaces" through folding operations); *languages abstracting communication and networks*, like TOTA [29] and Linda-$\sigma\tau$ [40] (supporting diffusion and aggregation of tuples on a network of agents), Logical Neighbourhoods [31] (supporting virtual connectivity), and SpatialViews [34] (abstracting a network into *spatial views* that can be iterated on to visit nodes and request services); and *macro-programming languages*, like *SpaceTime Oriented Programming (STOP)* [43] (providing abstractions to support collection and processing of past or future network data in arbitrary spatio-temporal resolutions) and *Regiment* [33] (modelling network state and regions as spatially distributed, time-varying signals). Aggregate Computing belongs to the class of

so-called general-purpose spatial computing languages, all addressing the problem of engineering distributed (or parallel) computing by providing mechanisms to manipulate data structures diffused in space and evolving in time. Other notable examples include the SDEF programming system inspired by systolic computing [23], and topological computing with MGS [25]. They typically provide specific abstractions that significantly differ from that of computational fields: for instance, MGS defines computations over manifolds, the goal of which is to alter the manifold itself as a way to represent input-output transformation.

3 Featherweight ScaFi: a Core Calculus for ScaFi

In this section, we present Featherweight ScaFi (FScaFi), a minimal core calculus that models the aggregate computing aspects of ScaFi—much as FJ [27] models the object-oriented aspects of Java.

In the aggregate computing model, devices undergo computation in rounds. When a round starts, the device gathers information about messages received from neighbours (only the last message from each neighbour is actually considered), performs an evaluation of the program, and finally emits a message to all neighbours with information about the outcome of computation. The scheduling policy of such rounds is abstracted in this formalisation, though it is typically considered fair and non-synchronous.

FScaFi is a core subset of ScaFi, *strictly retaining its syntax* (with the exception of typing annotations, which are not here presented). The syntax of FScaFi is given in Figure 1. Following [27], the overbar notation denotes metavariables over sequences and the empty sequence is denoted by $\bullet$; e.g., for expressions, we let $\overline{\texttt{e}}$ range over sequences of expressions, written $\texttt{e}_1$, $\texttt{e}_2$, ... $\texttt{e}_n$ ($n \geq 0$). FScaFi focusses on aggregate programming constructs. In particular:

- it neglects the many orthogonal Scala features that one can use (object-oriented constructs, and the like), and
- it is parametric in the built-in data constructors and functions.

Note that – apart from specific Scala syntax – the examples of ScaFi code given in Section 4 are actually examples of FScaFi code. In particular, in order to turn ScaFi functions (such as `foldhoodPlus`, `gradient` and `branch`—covered in Section 4) into FScaFi functions, it is enough to drop type annotations and default parameters.

A program `P` consists of a sequence $\overline{\texttt{F}}$ of function declarations and a main expression `e`. A function declaration `F` defines a (possibly recursive) function; it consists of a name `d`, $n \geq 0$ variable names $\overline{\texttt{x}}$ representing the formal parameters, and an expression `e` representing the body of the function.

Expressions `e` are the main entities of the calculus, modelling a whole field computation. An expression can be: a variable `x`, used as function formal parameter; a value `v`; an anonymous function $(\overline{\texttt{x}})$ $\overset{\tau}{\texttt{=>}}$ `@@{e}` (where $\overline{\texttt{x}}$ are the

$\texttt{P} ::= \overline{\texttt{F}}\ \texttt{e}$	program
$\texttt{F} ::= \texttt{def d}(\overline{\texttt{x}}) = \texttt{@@\{e\}}$	function declaration
$\texttt{e} ::= \texttt{x} \mid \texttt{v} \mid (\overline{\texttt{x}}) \stackrel{\tau}{=>} \texttt{@@\{e\}} \mid \texttt{e}(\overline{\texttt{e}})$ $\mid \texttt{rep(e)\{e\}} \mid \texttt{nbr\{e\}} \mid \texttt{foldhood(e)(e)\{e\}}$	expression
$\texttt{v} ::= \texttt{c}(\overline{\texttt{v}}) \mid \texttt{f}$	value
$\texttt{f} ::= \texttt{b} \mid \texttt{d} \mid (\overline{\texttt{x}}) \stackrel{\tau}{=>} \texttt{@@\{e\}}$	function value

Fig. 1: Syntax of FScaFi.

formal parameters, `e` is the body, and τ is a *tag*); a function call $\texttt{e}(\overline{\texttt{e}})$; a `rep`-expression `rep(e){e}`, modelling time evolution; an `nbr`-expression `nbr{e}`, modelling neighbourhood interaction; or a `foldhood`-expression `foldhood(e)(e){e}` which combines values obtained from neighbours.

Tags τ of anonymous functions $(\overline{\texttt{x}}) \stackrel{\tau}{=>} \texttt{@@\{e\}}$ do not occur in source programs: when the evaluation starts each anonymous function expression $(\overline{\texttt{x}})$ `=> @@{e}` occurring in the program is given a distinguished tag τ—for instance, two occurrences of the same anonymous function expression get different tags. In the following we will use the phrase *name of a function* to refer both to the tag of an anonymous function, or to the name of a built-in or declared function. As we will see below, names are used to define function equality.

The set of the *free variables* of an expression `e`, denoted by $\mathbf{FV}(\texttt{e})$, is defined as usual (the only binding construct is $(\overline{\texttt{x}}) \stackrel{\tau}{=>} \texttt{@@\{e\}}$). An expression `e` is *closed* if $\mathbf{FV}(\texttt{e}) = \bullet$. The main expression of any program must be closed.

A value can be either a *data value* $\texttt{c}(\overline{\texttt{v}})$ or a *functional value* `f`. A data value consists of a *data constructor* `c` of some arity $m \geq 0$ applied to a sequence of m data values $\overline{\texttt{v}} = \texttt{v}_1, ..., \texttt{v}_m$. A data value $\texttt{c}(\texttt{v}_1, ..., \texttt{v}_m)$ is written `c` when $m = 0$. Examples of data values are: the Booleans `True` and `False`, numbers, pairs (like $\texttt{Pair}(\texttt{True}, \texttt{Pair}(5, 7))$) and lists (like $\texttt{Cons}(3, \texttt{Cons}(4, \texttt{Null}))$).

Functional values `f` comprise:

- declared function names `d`;
- closed anonymous function expressions $(\overline{\texttt{x}}) \stackrel{\tau}{=>} \texttt{@@\{e\}}$ (i.e., such that $\mathbf{FV}(\texttt{e}) \subseteq \{\overline{\texttt{x}}\}$);
- built-in functions `b`, which can in turn be:
  - *pure operators* `o`, such as functions for building and decomposing pairs (`pair`, `fst`, `snd`) and lists (`cons`, `head`, `tail`), the equality function (=), mathematical and logical functions (`+`, `&&`, ...), and so on;
  - *sensors* `s`, which depend on the current environmental conditions of the computing device δ, such as a `temperature` sensor—modelling construct `sense` in ScaFi;
  - *relational sensors* `r`, modelling construct `nbrvar` in ScaFi, which in addition depend also on a specific neighbour device δ' (e.g., `nbrRange`, which measures the distance with a neighbour device).

In case `e` is a binary built-in function `b`, we shall write $\texttt{e}_1$ `b` $\texttt{e}_2$ for the function call $\texttt{b}(\texttt{e}_1, \texttt{e}_2)$ whenever convenient for readability of the whole expression in which it is contained.

The key constructs of the calculus are:

- *Function call:* $\texttt{e}(\texttt{e}_1, \ldots, \texttt{e}_n)$ is the main construct of the language. The function call evaluates to the result of applying the function value `f` produced by the evaluation of `e` to the value of the parameters $\texttt{e}_1, \ldots, \texttt{e}_n$ *relatively to the aligned neighbours*, that is, relatively to the neighbours that in their last execution round have evaluated `e` to a function value with the *same name* of `f`. For instance, suppose to have defined a function `def` $plus(a, b) =$ `@@{a + b}`; then, function call $plus(5, 2)$ yields a field that is 7 in every point of space and time (i.e., the expression evaluates to 7 in each round of every device).
- *Time evolution:* $\texttt{rep}(\texttt{e}_1)\{\texttt{e}_2\}$ is a construct for dynamically changing fields through the "repeated" application of the functional expression $\texttt{e}_2$. At the first computation round (or, more precisely, when no previous state is available—e.g., initially or at re-entrance after state was cleared out due to branching), $\texttt{e}_2$ is applied to $\texttt{e}_1$, then at each other step it is applied to the value obtained at the previous step. For instance, $\texttt{rep}(0)\{(\texttt{x}) \Rightarrow \texttt{@@}\{\texttt{x} + 1\}\}$ counts how many rounds each device has computed (from the beginning, or more generally, since that piece of state was missing). Another example is an expression $\texttt{snd}(\texttt{rep}(\texttt{Pair}(\texttt{x}, \texttt{False}))\{(\texttt{x}_R) \Rightarrow \texttt{@@}\{\texttt{Pair}(\texttt{x}, \texttt{x} == \texttt{fst}(\texttt{x}_\text{R}))\}\})$ that evaluates to `True` when some value `x` changes w.r.t. the previous round; it is common to use tuples when dealing with multiple pieces of state/result.
- *Neighbourhood interaction:* $\texttt{foldhood}(\texttt{e}_1)(\texttt{e}_2)\{\texttt{e}_3\}$ and $\texttt{nbr}\{\texttt{e}\}$ model device-to-device interaction. The `foldhood` construct evaluates expression $\texttt{e}_3$ *against every aligned neighbour*[6] (including the device itself), then aggregates the values collected through $\texttt{e}_2$ together with the initial value $\texttt{e}_1$. The `nbr` construct tags expressions `e` signalling that (when evaluated against a neighbour) the value of `e` has to be gathered from neighbours (and not directly evaluated). Such behaviour is implemented via a conceptual broadcast of the values evaluated for `e`. Subexpressions of $\texttt{e}_3$ not containing `nbr` are *not* gathered from neighbours instead.
  As an example, consider the expression

$$\texttt{foldhood}(2)(+)\{\texttt{min}(\texttt{nbr}\{\texttt{temperature}()\}, \texttt{temperature}())\}$$

  evaluated in device δ_1 (in which `temperature()` = 10) with neighbours δ_2 and δ_3 (in which `temperature()` gave 15 and 5 in their last evaluation round, orderly). The result of the expression is then computed adding 2, $\texttt{min}(10, 10)$, $\texttt{min}(15, 10)$ and $\texttt{min}(5, 10)$ for a final value of 27.

[6] This is where FScaFi differs from classical field calculus, where instead neighbouring fields are explicitly manipulated.

Note that, according to the explanation given above, calling a declared or anonymous function acts as a branch, with each function in the range applied only on the subspace of devices holding a function with the same tag. In fact, a branching construct `branch`(e_1)\{e_2\}\{e_3\} (which computes e_2 or e_3 depending on the value of e_1) can be defined through function application as `mux`(e_1, () => `@@`\{e_2\}, () => `@@`\{e_3\})(), where `mux` is a built-in function selecting among its second and third argument depending on the value of the first.

Notice that the semantics of this language is compositional and message exchanges are performed under the hood by `nbr` constructs within a `foldhood`; with an automatic matching of each message from a neighbour to a specific `nbr` construct, determined through a process called *alignment* [5]. Basically, each `nbr` construct produces an "export" (i.e., a data value to be sent to neighbours) tagged with the coordinates of the node in the evaluation tree (i.e., the structure arising from the dynamic unfolding of the main expression evaluation) up to that construct. All exports are gathered together into a message which is broadcast to neighbours, and which can be modelled as a *value tree*: an ordered tree of values obtained during evaluation of each sub-expression of the program. The alignment mechanism then ensures that each `nbr` is matched with corresponding `nbr` reached by neighbours with an identical path in the evaluation tree.

4 Showcasing FScaFi: Programming Examples

In this section, we provide examples of FScaFi programs, showing how to represent and manipulate fields to implement collective adaptive functionality, using the ScaFi syntax.

4.1 Scala Syntax

In ScaFi, the FScaFi constructs introduced in Section 3 are represented as object-oriented methods through the following Scala trait (interface):

```
trait Constructs {
  def rep[A](init: => A)(fun: (A) => A): A
  def foldhood[A](init: => A)(aggr: (A, A) => A)(expr: => A): A
  def nbr[A](expr: => A): A
  def @@[A](b: => A): A
}
```

This is mostly a straightforward Scala encoding of the syntax of Figure 1. The main different is given by the presence of *typing information* and, in particular, the use of *by-name parameters*, of type $=>T$, which provide syntactic sugar for 0-ary functions: these enable to capture expressions at the call site, pass them unevaluated to the method, and evaluate them lazily every time the parameter is used. This turns out very useful to implement the FScaFi semantics while providing a very lightweight syntax for the DSL. Moreover, note that method signatures do not include field-like type constructors: in fact, in FScaFi, fields are not reified explicitly but only exist at the semantic level.

4.2 Programming Examples

When thinking at field programs, one can adopt two useful viewpoints: the *local* viewpoint, typically useful when reasoning about low-level aspects of field computations, which considers a field expression as the computation carried on by a specific individual device; and the *global* viewpoint, typically more useful when focussing on higher-level composition of field computations, which regards a specification at the aggregate level, as a whole spatio-temporal computation evolving a field. So, an expression (e.g., `1+3` of type `Int`) can represent the outcome of execution of a computation locally (`4`), or globally as the program producing a field (a field of `4`s). Note, however, that the global field is not accessed computationally: a local computation will only access a *neighbouring* field (which is actually a *view*, given by the messages received from neighbours, of the actual, asynchronously evolving field).

In the following, we incrementally describe the constructs introduced in Section 3 and the design of higher-level building blocks of collective adaptive behaviour through examples. In ScaFi, a usual literal such as, for instance, tuple

```
("hello", 7.7, true)
```

is to be seen as a *constant* (i.e., not changing over time) and *uniform* (i.e., not changing across space) field holding the corresponding local value at any point of the space-time domain. By analogy, an expression such as

```
1 + 2
```

denotes a global expression where a field of `1`s and a field of `2`s are summed together through the field operator +, which works like a point-wise application of its local counterpart. Indeed, literal + can also be thought of as representing a constant, uniform field of (binary) functions, and function application can be viewed as a global operation that applies a function field to its argument fields.

A constant field does not need to be uniform. For instance, given a static network of devices, then

```
mid()
```

denotes the field of device identifiers, which does not change across time but does vary in space. On the other hand, expression

```
sense[Double]("temperature") // type can be omitted if can be inferred
```

is used to represent a field of temperatures (as obtained by collectively querying the local temperature sensors over space and time), which is in general non-constant and non-uniform.

Fields changing over time can also be programmatically defined by the `rep` operator; for instance, expression

```
// Initially 0; state is incremented at each round
rep(0){ x => x + 1 } // Equally expressed in Scala as: rep(0)(_ + 1)
```

counts how many rounds each device has executed: it is still a non-uniform field since the update phase and frequency of the devices may vary both between devices and across time for a given device.

Folding can be used to trigger the important concept of neighbour-dependent expression. As a simple initial example, expression

```
foldhood(0)(_ + _){ 1 }
```

counts the number of neighbours at each device (possibly changing over time if the network topology is dynamic). Note that folding collects the result of the evaluation of 1 against all neighbours, which simply yields 1, so the effect is merely the addition of 1 for each existing neighbour.

The key way to define truly neighbour-dependent expressions is by the `nbr` construct, which enables to "look around" just one step beyond a given locality. Expression

```
foldhood(0)(_+_){nbr{sense[Double]("temperature")}} / foldhood(0)(_+_){1}
```

evaluates to the field of average temperature that each device can perceive in its neighbourhood. The numerator sums temperatures sensed by neighbours (or, analogously, it sums the neighbour evaluation of the temperature sensor query expression), while the denominator counts neighbours as described above. As another example, the following expression denotes a Boolean field of warnings:

```
val warningTh: Double = 42.0 // temperature threshold for warning
foldhood(false)(_ || _){ nbr { sense[Double]("temperature") } > warningTh }
```

This is locally `true` if any neighbour perceives a temperature higher than some topical threshold. Notice that by moving the comparison into the `nbr` block,

```
foldhood(false)(_ || _){ nbr { sense[Double]("temperature") > warningTh } }
```

then the decision about the threshold (i.e., the responsibility of determining when a temperature is dangerous) is transferred to the neighbours, and hence warnings get blindly extended by 1-hop. Of course, provided `warningTh` is uniform, the result would be the same in this case.

Functions can be defined to capture and give a name to common field computation idioms, patterns, and domain-specific operations. For instance, by assuming a `mux` function that implements a strictly-evaluated version of `if`:

```
def mux[A, B<:A, C<:A](cond: Boolean)(th: B)(el: C): A = if(cond) th else el
```

A variation of `foldhood`, called `foldhoodPlus`[7], which does not take "self" (the current device) into account, can be implemented as follows:

```
def foldhoodPlus[A](init: => A)(aggr: (A, A) => A)(expr: => A): A =
  foldhood(init)(aggr)(mux(mid==nbr{mid}){ init }{ expr })
```

[7] The "Plus" suffix is to mimic the mathematical syntax R^+ of the transitive closure of a (neighbouring) relation R.

Notice that the identity `init` is used when considering a neighbour device whose identifier (`nbr{mid}`) is the same as that of the current device (`mid`). As another example, one can give a label to particular sensor queries, such as:

```
def temperature = sense[Double]("temperature")
def nbrRange = nbrvar[Double]("nbr-range")
```

The second case uses construct `nbrvar`, which is a neighbouring sensor query operator providing, for each device, a sensor value for each corresponding neighbour: e.g., for `nbrRange`, the output value is a floating-point number expressing the estimation of the distance from the currently executing device to that neighbour—so, it is usually adopted as a metric for "spatial algorithms". Based on the above basic expressions, one can define a rather versatile and reusable building block of Aggregate Programming, called *gradient* [28, 9, 4]. A gradient (see Figure 2) is a numerical field expressing the minimum distance (according to a certain `metric`) from any device to `source` devices; it is also interpretable as a surface whose "slope" is directed towards a given source. In ScaFi, it can be programmed as follows:

```
def gradient(source: Boolean, metric: () => Double = nbrRange): Double =
  rep(Double.PositiveInfinity){ distance =>
    mux(source) { 0.0 }{
      foldhoodPlus(Double.PositiveInfinity)
                  (Math.min(_,_)) { nbr{distance} + metric }
  } }
```

The `rep` construct allows one to keep track of the distances across rounds of computations: source devices are at a null distance from themselves, and the other devices take the minimum value among those of neighbours increased by the corresponding estimated distances as given by `metric`—defaulting to `nbrRange`. Notice that `foldhoodPlus` (i.e., a version of `foldhood` that does not consider the device itself) must be used to prevent devices from getting stuck to low values because of self-messages (as it would happen when a source node gets deactivated): with it, gradients dynamically adapt to changes in network topology or position/number of sources, i.e., it is self-stabilising [36].

Another key operation on fields is splitting computation into completely separate parts or sub-computations executed in isolated space-time regions. An example is computing a `gradient` in a space that includes obstacle nodes so that gradient slopes circumvent the obstacles. The solution to the problem needs to leverage aggregate functions, and their ability of acting as units of alignment. That is, we can use a different aggregate function for normal and obstacle nodes:

```
(mux(isObstacle){ () => @@{ Double.PositiveInfinity } }
                { () => @@{ gradient(isSource) } }
)()
```

Calling such functions effectively restricts the domain to the set of devices executing them, thanks to the space-time branching enacted by construct `@@` wrapping *the bodies* of the corresponding literal functions; by calling them exclusively

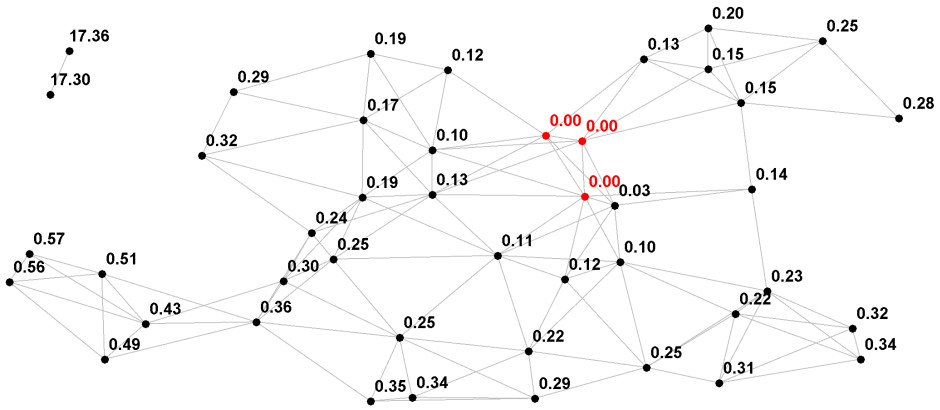

Fig. 2: Snapshot of a gradient field from a simulation in ScaFi. The red nodes are the sources of the gradient. The nodes at the top-left have parted from the network and their values increase unboundly. The gray lines represent device connectivity according to a proximity-based neighbouring relationship.

in any device, the system gets partitioned into two sub-systems, each one executing a different sub-computation. For convenience, ScaFi provides as built-in function, called `branch`, defined as:

```
def branch[A](cond: => Boolean)(th: => A)(el: => A): A =
  mux(cond)(() => @@{ th })(() => @@{ el })()
```

With it, a gradient overcoming an obstacle is properly written as

```
branch(isObstacle){ Double.PositiveInfinity }{ gradient(isSource) } // correct
```

which is cleaner and hides some complexity while better communicating the intent: branching computation. Generally, notation `@@` has to be used for bodies of literal functions that include aggregate behaviour, i.e., functions which (directly or indirectly) call methods of the `Constructs` trait—other uses have no effects on the result of computation. We remark that the above field calculus expression (gradient avoiding obstacles) effectively creates a distributed data structure that is rigorously self-adaptive [36]: independently of the shape and dynamics of obstacle area(s), source area(s), metric and network structure, it will continuously bring about formation of the correct gradient, until eventually stabilising to it. For instance, it could be used in a wireless sensor network scenario to let devices equipped with sensors transmit local perception on a hop-by-hop basis until all information reaches the gradient source. Generally, gradients can be used as building block for more complex behaviour, to which the self-adaption properties will be transferred, by simple functional composition.

An example of more complex behaviour is the *self-healing channel* [38], i.e., the field of Booleans that self-stabilises to a `true` value in the devices belonging

to the minimal path connecting source with target devices. This functionality can be implemented as follows:

```
def channel(source: Boolean, target: Boolean, width: Double): Boolean =
  gradient(source) + gradient(target) <=
    distanceBetween(source, target) + width
```

i.e., by applying the triangle inequality property (with some tolerance as captured by parameter `width`), and exploiting a block `distanceBetween` that calculates the distance between `source` and `target` (e.g., using a `gradient`) and broadcasts that value to the whole network (e.g., by gossiping or along another gradient).

5 Conclusion and Future Work

Aggregate Computing is a recent paradigm for "holistically" engineering CASs and smart situated ecosystems, which aims to exploit, both functionally and operationally, the increasing computational capabilities of our environments—as fostered by driver scenarios like IoT, CPS, and smart cities. It formally builds on computational fields and corresponding calculi to functionally compose macro behavioural specifications that capture, in a declarative way, the adaptive logic for turning local activity into global, resilient behaviour. In this paper, we have introduced FScaFi, a core calculus that captures the essential features of ScaFi, a recently developed Scala-internal aggregate programming DSL. In particular, it leverages a novel notion of "computation against a neighbour" which enabled seamless integration in the Scala language and type system.

In future work, we will formalise the FScaFi semantics (informally sketched in this paper), its properties, and relation with the field calculus—mainly aimed at proving analogous properties such as those in [19, 12, 36, 3]. Additionally, work on ScaFi is part of the research agenda for Aggregate Computing as comprehensively covered in [37], which includes the study of dynamic field computations, or aggregate processes [17] as well as the design and implementation of aggregate computing runtime platforms [39, 16]. Together, they could lead to the emergence of a new platform for large-scale distributed systems deployment over phyisical environments (such as the IoT), whereby distributed computations can be dynamically injected, executed in a distributed way, and cooperate and compete with each other to realise an ecosystem of adaptive services—developing on the the vision of, e.g., [30].

References

1. Alrahman, Y.A., De Nicola, R., Loreti, M., Tiezzi, F., Vigo, R.: A calculus for attribute-based communication. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. pp. 1840–1845 (2015)
2. Anderson, S., Bredeche, N., Eiben, A., Kampis, G., van Steen, M.: Adaptive collective systems: herding black sheep (2013)

3. Audrito, G., Beal, J., Damiani, F., Viroli, M.: Space-time universality of field calculus. In: Serugendo, G.D.M., Loreti, M. (eds.) Coordination Models and Languages. Lecture Notes in Computer Science, vol. 10852, pp. 1–20. Springer (2018). https://doi.org/10.1007/978-3-319-92408-3_1
4. Audrito, G., Casadei, R., Damiani, F., Viroli, M.: Compositional blocks for optimal self-healing gradients. In: 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO. pp. 91–100. IEEE Computer Society (2017). https://doi.org/10.1109/SASO.2017.18
5. Audrito, G., Damiani, F., Viroli, M., Casadei, R.: Run-time management of computation domains in field calculus. In: Foundations and Applications of Self* Systems, IEEE International Workshops on. pp. 192–197. IEEE (2016). https://doi.org/10.1109/FAS-W.2016.50
6. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Transactions on Computational Logic **20**(1), 5:1–5:55 (2019). https://doi.org/10.1145/3285956
7. Baude, F., Henrio, L., Ruz, C.: Programming distributed and adaptable autonomous components—the GCM/ProActive framework. Software: Practice and Experience **45**(9), 1189–1227 (2015). https://doi.org/10.1002/spe.2270
8. Beal, J., Bachrach, J.: Infrastructure for engineered emergence in sensor/actuator networks. IEEE Intelligent Systems **21**, 10–19 (March/April 2006)
9. Beal, J., Bachrach, J., Vickery, D., Tobenkin, M.: Fast self-healing gradients. In: Proceedings of ACM SAC 2008. pp. 1969–1975. ACM (2008)
10. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Mernik, M. (ed.) Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, chap. 16, pp. 436–501. IGI Global (2013). https://doi.org/10.4018/978-1-4666-2092-6.ch016
11. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. IEEE Computer **48**(9) (2015)
12. Beal, J., Viroli, M., Pianini, D., Damiani, F.: Self-adaptation to device distribution in the Internet of Things. ACM Transaction on Autonomous and Adaptive Systems **12**(3), 12:1–12:29 (Sep 2017). https://doi.org/10.1145/3105758
13. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt (2016)
14. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: Deeco: an ensemble-based component system. In: Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering. pp. 81–90. ACM (2013). https://doi.org/10.1145/2465449.2465462
15. Casadei, R., Pianini, D., Viroli, M.: Simulating large-scale aggregate mass with alchemist and scala. In: Computer Science and Information Systems (FedCSIS), 2016 Federated Conference on. pp. 1495–1504. IEEE (2016)
16. Casadei, R., Viroli, M.: Programming actor-based collective adaptive systems. In: Programming with Actors: State-of-the-Art and Research Perspectives, LNCS, vol. 10789, pp. 94–122. Springer (2018). https://doi.org/10.1007/978-3-030-00302-9_4
17. Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F.: Aggregate processes in field calculus. In: Riis Nielson, H., Tuosto, E. (eds.) Coordination Models and Languages. pp. 200–217. Springer (2019). https://doi.org/10.1007/978-3-030-22397-7_12
18. Coore, D.: Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer. Ph.D. thesis, MIT (1999)

19. Damiani, F., Viroli, M.: Type-based self-stabilisation for computational fields. Logical Methods in Computer Science **11**(4) (2015). https://doi.org/10.2168/LMCS-11(4:21)2015
20. De Nicola, R., Duong, T., Inverso, O., Trubiani, C.: Aerlang: empowering erlang with attribute-based communication. Science of Computer Programming **168**, 71–93 (2018)
21. De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **9**(2), 7:1–7:29 (2014). https://doi.org/10.1145/2619998
22. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for java. In: ACM SIGPLAN Notices. vol. 48, pp. 112–121. ACM (2012)
23. Engstrom, B.R., Cappello, P.R.: The SDEF programming system. Journal of Parallel and Distributed Computing **7**(2), 201 – 231 (1989)
24. Ghosh, D.: DSL for the uninitiated. Commun. ACM **54**(7), 44–50 (2011). https://doi.org/10.1145/1965724.1965740
25. Giavitto, J.L., Michel, O., Cohen, J., Spicher, A.: Computations in space and space in computations. In: Unconventional Programming Paradigms, Lecture Notes in Computer Science, vol. 3566, pp. 137–152. Springer, Berlin (2005)
26. Hennicker, R., Klarl, A.: Foundations for ensemble modeling—the Helena approach. In: Specification, Algebra, and Software, pp. 359–381. Springer (2014). https://doi.org/10.1007/978-3-642-54624-2_18
27. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems **23**(3) (2001)
28. Lin, F.C.H., Keller, R.M.: The gradient model load balancing method. IEEE Trans. Softw. Eng. **13**(1), 32–38 (1987). https://doi.org/10.1109/TSE.1987.232563
29. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The tota approach. ACM Trans. on Software Engineering Methodologies **18**(4), 1–56 (2009). https://doi.org/10.1145/1538942.1538945
30. Montagna, S., Viroli, M., Fernandez-Marquez, J.L., Di Marzo Serugendo, G., Zambonelli, F.: Injecting self-organisation into pervasive service ecosystems. Mobile Networks and Applications **18**(3), 398–412 (2013). https://doi.org/10.1007/s11036-012-0411-1
31. Mottola, L., Picco, G.P.: Logical neighborhoods: A programming abstraction for wireless sensor networks. In: DCOSS. Lecture Notes in Computer Science, vol. 4026, pp. 150–168. Springer (2006)
32. Nagpal, R.: Programmable pattern-formation and scale-independence. In: Unifying themes in complex systems IV, pp. 275–282. Springer (2008)
33. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: Workshop on Data Management for Sensor Networks. pp. 78–87 (2004)
34. Ni, Y., Kremer, U., Stere, A., Iftode, L.: Programming ad-hoc networks of mobile and resource-constrained devices. ACM SIGPLAN Notices **40**(6), 249–260 (2005)
35. Pianini, D., Viroli, M., Beal, J.: Protelis: Practical aggregate programming. In: ACM Symposium on Applied Computing 2015. pp. 1846–1853 (April 2015)
36. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. **28**(2), 16:1–16:28 (Mar 2018). https://doi.org/10.1145/3177774

37. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. Journal of Logical and Algebraic Methods in Programming **109**, 100486 (2019). https://doi.org/10.1016/j.jlamp.2019.100486
38. Viroli, M., Beal, J., Damiani, F., Pianini, D.: Efficient engineering of complex self-organising systems by self-stabilising fields. In: Self-Adaptive and Self-Organizing Systems (SASO), 2015 IEEE 9th International Conference on. pp. 81–90. IEEE (Sept 2015). https://doi.org/10.1109/SASO.2015.16
39. Viroli, M., Casadei, R., Pianini, D.: On execution platforms for large-scale aggregate computing. In: Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct. pp. 1321–1326. ACM (2016)
40. Viroli, M., Pianini, D., Beal, J.: Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In: Proceedings of Coordination 2012, Lecture Notes in Computer Science, vol. 7274, pp. 212–229. Springer (2012)
41. Vlissides, J.M.: Pattern hatching: design patterns applied. Addison-Wesley Reading (1998)
42. Voelter, M.: DSL Engineering: Designing, Implementing and Using Domain-specific Languages. CreateSpace Independent Publishing Platform (2013)
43. Wada, H., Boonma, P., Suzuki, J.: A spacetime oriented macroprogramming paradigm for push-pull hybrid sensor networking. In: 2007 16th International Conference on Computer Communications and Networks. pp. 868–875. IEEE (2007)