COMPUTATION AGAINST A NEIGHBOUR:
ADDRESSING LARGE-SCALE DISTRIBUTION AND ADAPTIVITY
WITH FUNCTIONAL PROGRAMMING AND SCALA

GIORGIO AUDRITO ●[a], ROBERTO CASADEI ●[b], FERRUCCIO DAMIANI ●[a],
AND MIRKO VIROLI ●[b]

[a] Università degli Studi di Torino, Corso Svizzera, 185, Turin, Italy
e-mail address: giorgio.audrito@unito.it, ferruccio.damiani@unito.it

[b] Alma Mater Studiorum–Università di Bologna, Via Cesare Pavese, 50, Cesena, Italy
e-mail address: roby.casadei@unibo.it, mirko.viroli@unibo.it

ABSTRACT. Recent works in contexts like the Internet of Things (IoT) and large-scale Cyber-Physical Systems (CPS) propose the idea of programming distributed systems by focussing on their global behaviour across space and time. In this view, a potentially vast and heterogeneous set of devices is considered as an "aggregate" to be programmed as a whole, while abstracting away the details of individual behaviour and exchange of messages, which are expressed declaratively. One such a paradigm, known as aggregate programming, builds on computational models inspired by field-based coordination. Existing models such as the field calculus capture interaction with neighbours by a so-called "neighbouring field" (a map from neighbours to values). This requires ad-hoc mechanisms to smoothly compose with standard values, thus complicating programming and introducing clutter in aggregate programs, libraries and domain-specific languages (DSLs).

To address this key issue we introduce the novel notion of "computation against a neighbour", whereby the evaluation of certain subexpressions of the aggregate program are affected by recent corresponding evaluations in neighbours. We capture this notion in the *neighbours calculus* (NC), a new field calculus variant which is shown to smoothly support declarative specification of interaction with neighbours, and correspondingly facilitate the embedding of field computations as internal DSLs in common general-purpose programming languages—as exemplified by a Scala implementation, called ScaFi. This paper formalises NC, thoroughly compares it with respect to the classic field calculus, and shows its expressiveness by means of a case study in edge computing, developed in ScaFi.

1. INTRODUCTION

Pervasive computing, Internet of Things (IoT), Cyber-Physical Systems (CPS), Smart Cities and related initiatives, all point out a trend in informatics envisioning a future where computation is fully pervasive and ubiquitous, and is 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 increasingly emerging: a large-scale network

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.46298/LMCS-19(1:6)2023

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 single "distributed machine". These kinds of systems are sometimes referred to as *Collective Adaptive Systems (CAS)* [ABE+13, NJW20], 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—whether they are changes/faults in the computational environment or pertain unexpected interaction with humans or other systems.

Aggregate Computing [BPV15] is a macro-approach [Cas22a] to CAS engineering [NJW20] where a network of devices operating at asynchronous sense-compute-interact rounds is programmed through a single macro-level program, called an *aggregate program*. An aggregate program expresses both computation and interaction, and its repeated execution by devices on their local contexts promotes the unfolding of collective adaptive behaviour. This approach is especially suitable to problems and application domains such as crowd engineering, complex situated coordination, robot/UAV swarms, smart ecosystems and the like [VBD+19]. One fundamental enabling abstraction for specifying the dynamics of situated collectives is that of a *computational field* (or simply, field) [AVD+19, BB06, MZ09]: a distributed data structure that maps devices to computational objects across time. Accordingly, *Aggregate Programming* is about describing 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.

Aggregate computing is part of *field-based coordination* [VBD+19]. The core formal language capturing the essence of aggregate computing using fields is the *field calculus* [AVD+19]. The field calculus can be seen as a particular kind of *context-oriented programming* [SGP12] approach with *multiple local contexts* where behavioural variations are determined through branching conditions on field values [CPSV19]. The field calculus and its computational model also provide an expressive formal framework for supporting both the investigation of properties about distributed computations (like self-stabilisation [VAB+18], density-independence [BVPD17], real-time guarantees [ADVB18], universality [ABDV18], time-fluidity [PCV+21], and deployment flexibility [CFP+22]) and novel constructs (like aggregate processes [CVA+19] and the *share* construct [ABD+20]). The field calculus had been implemented by Domain Specific Languages (DSLs) such as Proto (a Scheme external DSL) [BB06], Protelis (a Java external DSL) [PVB15], FCPP (a C++ internal DSL) [Aud20]. These DSLs rely on the notion of "neighbouring value" (a map from neighbours to data values) to model device interaction. A neighbouring value is used to locally express the outcome of message reception from neighbours, and to manipulate such information to collect resulting values. As a consequence, these languages need two classes of operators and types: one for dealing with local values, and one for dealing with collection-like, neighbouring values. Managing and reconciling these two kinds of operations may tend to complicate aggregate programming, design of libraries, and language implementation. Existing DSLs deal with this issue in a variety of ways: relying on dynamic typing (as in Proto), using macros and meta-programming techniques to alleviate such issues to the user (as in FCPP), or requiring duplication of operation across local and neighbouring values (as in Protelis)—none of which is completely satisfactory.

In order to resolve the duality of local/neighbouring values, with the goal of *conceptual economy* and to promote *smooth embedding* of field computations in mainstream languages and programming practice, in this paper we introduce a novel notion of "computation

against a neighbour", used to express interaction in field-based coordination by entirely replacing the notion of neighbouring value. The key idea is to allow the evaluation of a certain sub-expression of the aggregate program to depend on a recent outcome of the evaluation of the same subexpression in a neighbour. Such a dependency is hence expressed fully declaratively, without escaping the functional paradigm of field-based computing, and is then internally implemented by asynchronous message exchange across neighbours. To present this mechanism and study its implications, in this paper we:

(1) define syntax, typing, operational semantics of a foundation calculus, called *neighbours calculus* (NC);
(2) investigate the properties of the NC and its relationship with the field calculus;
(3) show the advantages that NC brings in term of smooth embedding into Scala—host language chosen for its hybrid object-oriented/functional nature, flexibility, and suitability for internal DSL development [AHKY15, CRD$^+$17].

In particular, the last contribution is based on the implementation of the NC computational model in *ScaFi*[1] (*Sca*la *Fi*elds). ScaFi is a Scala [Slo08] aggregate programming toolkit comprising an internal DSL for aggregate programming, which has been used in a variety of applications [CVA$^+$21, CTVD19, CV19, CAV18]. Hence, in addition to formalising the "computation against a neighbour" mechanism, NC also serves as formalisation of the core of ScaFi.

    The remainder of this paper is structured as follows. Section 2 provides background information about Aggregate Computing and field computations, as well as motivation for this work. Section 3 describes syntax, typing, and operational semantics of NC. Section 4 provides a detailed account on the properties of NC, by comparison with the field calculus. Section 5 presents the ScaFi Scala-based NC implementation, and the key advantage of the approach over other implementations approaches for field calculi. Section 6 describes an edge computing case study built on NC/ScaFi. Finally, Section 7 summarises related works, and Section 8 ends up the paper with a wrap-up and discussion of future works.

2. Background and Motivation

In this section, we recap Aggregate Computing (Section 2.1) and its formal embodiment into a computational model based on a *field* abstraction (Section 2.2), as a response to the need of engineering the collective adaptive behaviour of large-scale distributed and physically situated systems. Then, we describe how Aggregate Computing can be used to express the paradigmatic example of self-healing channels (Section 2.3), to give an early idea of how the approach works and its main benefits (e.g., composability). Finally, we motivate the work developed in this paper in terms of goals, decisions, and technical challenges (Section 2.4).

2.1. **Aggregate Computing.** Aggregate Computing is an approach to CAS development that abstracts from the traditional, device-centric viewpoint (where the programs describe what a device should do) in favour of a "holistic stance" where the target of design/development is the whole collection of situated and interacting devices that compose the system, seen as a single programmable, distributed, computational body [BPV15]. The core idea is to provide a single specification – which we call an *aggregate program* – of how the system globally behaves. This is done without even mentioning the existence of individuals, and

---

[1]`https://github.com/scafi/scafi`

hence independently of the shape and size of the set of devices: it is "under-the-hood" that the local behaviours to be executed by the individual devices are derived. This approach may be labelled as a *macro-to-micro* one, to differentiate it from the more classical (micro-to-macro) approach where the components are individually implemented in order to produce the intended system-level behaviour by more or less "controlled" emergence. Hence, from the programming perspective, the key advantage is the ability of *declaratively* expressing the logic of an ensemble, avoiding to directly solve the generally intractable local-to-global mapping problem and clearly *separating the concerns* of overall aggregate behaviour from the device-specific ones.

The general idea of programming at the macro level dates back to works such as [NW04, BMG08] in the context of wireless sensor networks, but it has recently received considerable renewed interest (see, e.g., the surveys [BDU+13, Cas22a] and the many related approaches mentioned in Section 7) with the emergence of the IoT. In this research context, the key contribution of Aggregate Computing lies in supporting *abstract and resilient composition of collective adaptive behaviours.* Since an aggregate program is expressed in a way that is independent of the actual number and dislocation of devices – and the resulting computation is a repetitive, gossip-like process of data distribution and computation – adaptation to changes and faults is inherent, and can be controlled by relying on specific programming patterns [VBDP15], often biologically inspired. Then, at this level, compositionality refers to the capability of combining behaviours in such a way that the result of the combination, and its properties, are in several cases fully predictable [BVPD17, VAB+18, VCP16a]. In particular, *self-stabilisation* is retained by a large class of aggregate programs: it makes an aggregate system able to tolerate circumscribed failures and react to disruptive changes in the environment in order to re-establish, after some transient adjusting phase, the proper operation.

2.2. **Computing with Fields.** The computational framework incarnating the ideas and goals of Aggregate Computing is based on the notion of *computational field* (or simply field) [VBD+19]. Intuitively, as shown in Figure 1, a computational field is an abstraction that represents a distributed data structure mapping points in space-time (the field *domain*) to values produced (by some device) as computation result at those points. Programming field computations hence encourages to reason in terms of evolving global structures that continuously express both the result of computation and the relationships between individuals.

Field computations generally comprise mechanisms for: *(i)* lifting standard values and computations, which are "local", to work as whole fields by flatly applying them to each space-time point of the domain; *(ii)* expressing the dynamics of fields, namely, how fields evolve over time; *(iii)* exchanging values across *neighbours*, such that information can flow beyond localities and interaction can unfold to realise complex patterns such as, e.g., outward propagation of local events; and *(iv)* branching fields into spatio-temporally-isolated sub-fields, in order to organise a computation into multiple parts co-existing in different space-time regions.

Inspired by minimal core calculi such as λ-calculus [Chu32] and FJ [IPW01], the above mechanisms have been formalised by the *field calculus* [AVD+19]. The field calculus relies on the basic functional programming model abstractions (i.e., first-class functions and functional composition) to support composability of distributed behaviour. In a nutshell, expressions denote whole fields; a specific construct, called `rep`, deals with field dynamics; another construct, called `nbr`, declaratively expresses interaction with neighbours; finally,
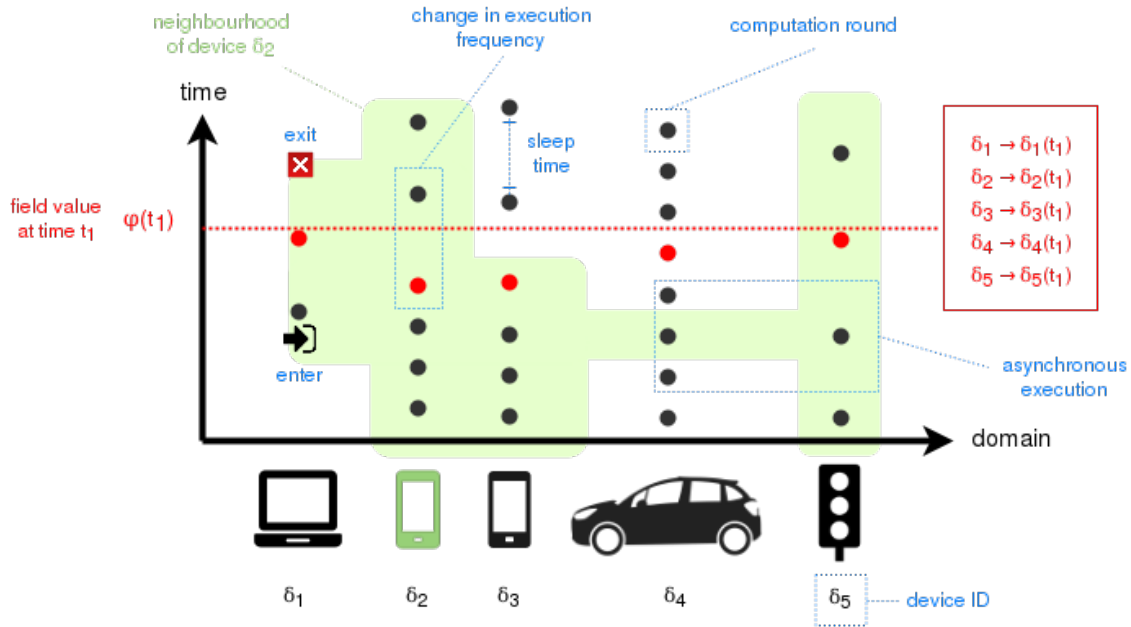
FIGURE 1. Dynamic view of a computational field: a field maps devices to values over time; devices can enter and exit a given *space-time domain* freely (openness); devices fire asynchronously (possibly at varying rates), execute a *computational round* upon fires producing the local value of the field, and then "sleep" in between fires. At a given time, a snapshot of the most-recent value at each device gives a field value (this view of an external, global observer is conceptually useful but global fields are not to be computationally manipulated in a direct fashion). A device can only directly communicate with a subset of other devices which is known as its *neighbourhood* (e.g., for δ_2, represented by the green area), which usually (but not necessarily) reflects spatial proximity in situated systems.

higher-order function call captures behaviour activation as well as *branching* (as a "field of functions" can be invoked through the *call* operator, space-time regions are naturally defined by where/when the same function is actually called).

As suggested in Figure 1, field computations assume a set of networked devices that run at asynchronous or partially synchronous *rounds of execution*. It is such an iterative execution that, combined with repetitive local sensing and device-to-device interaction, enables intrinsic adaptation to environmental perturbations and progressive steering of the system towards the desired states—much like in *swarm* and *computational collective intelligence* [BDT99, Szu01]. In each computation round, a device: *(i) determines its local context*, by retrieving any previous state and collecting sensor values as well as messages potentially received from its neighbours; *(ii) locally executes a field computation expression*, in a contextual fashion and according to the local semantics, producing an output and an *export* value which provides information for collective coordination; *(iii) shares the export with its neighbours*, through a conceptual broadcast; and finally *(iv) feeds actuators*, with the produced output. In other words, this execution model implements a sort of distributed,

continuous closed loop between field computation and the environment, where each device computes values that are contextually related with those computed by neighbours.

2.3. **Example: self-healing channel.** A paradigmatic example of what can be done using the Aggregate Computing approach is the *self-healing channel* (cf. Figure 2), i.e., the construction of a Boolean field that is true only in the set of devices that connect two endpoints of the system through a minimum path, and that automatically adapts to changes of the endpoints and the system connectivity network. Such a collective adaptive behaviour can be expressed through an aggregate program as follows:

```
// Definition
def channel(endpoint1, endpoint2) {
  gradient(endpoint1) + gradient(endpoint2) <= distanceBetween(endpoint1,endpoint2)
}
// Use
channel(isSource(), isDestination(), 10)
```

where `isSource` and `isDestination` are assumed to be sensors available in every device. The self-healing channel algorithm can be implemented through a function `channel` that reuses existing building blocks of collective behaviour, such as `gradient(s)` (computing the self-healing field of minimum distances from the source `s`) and `distanceBetween(a,b)` (spreading in the network the distance between `a` and `b`, which can be built over `gradient`s). It works by exploiting the triangle inequality principle: given a device at position p, the sum of the distance d_1 from `endpoint1` to p (computed by the first call to `gradient`) with the distance d_2 from `endpoint2` to p (computed by the second call to `gradient`) must be greater than (or equal to, if we consider degenerate triangles) the distance $d_{1\leftrightarrow2}$ from `endpoint1` to `endpoint2` (which is computed by the call to `distanceBetween`). So, the devices belonging to the channel are those for which $d_1 + d_2 \leq d_{1\leftrightarrow2}$, i.e., the devices that lie in the path from `endpoint1` to `endpoint2`. Then, the execution is decentralised, progressive, and open-ended. Given the round-based execution model mentioned previously, the expression corresponding to the `channel` function application is to be repeatedly evaluated by every device, so that up-to-date local information (perceived by sensors or obtained from recent messages from neighbours) is properly incorporated to progressively adjust the collective result. So, local changes (e.g., a device un/becomes a source, as perceived by `isSource`, or the distance to neighbours changes as a result of mobility, which is perceived internally in `gradient`s) will affect the local evaluation of the involved gradients, and such effects will propagate to neighbours, eventually affecting the rest of the network and the global channel result. Furthermore, it can be shown that this computation is self-stabilising [VAB+18], which means that it would eventually converge to the correct global result once the environmental inputs stop changing. It turns out that `gradient` can be implemented in many ways [ACDV17], but a simple implementation leverages a combination of `rep` and `nbr`—this will be described in detail and incrementally in Section 3 (cf. Examples 3.1 and 3.10). Notice that the use of `nbr` means that an evaluation of `gradient` will cause pertinent information to be exchanged with neighbours. The point of this example is not to fully understand the technical details involved, which will be clarified in Section 3, but to give the intuition of the compositionality of Aggregate Computing, where functions expressing collective adaptive behaviour can be combined to express more complex collective adaptive behaviour.
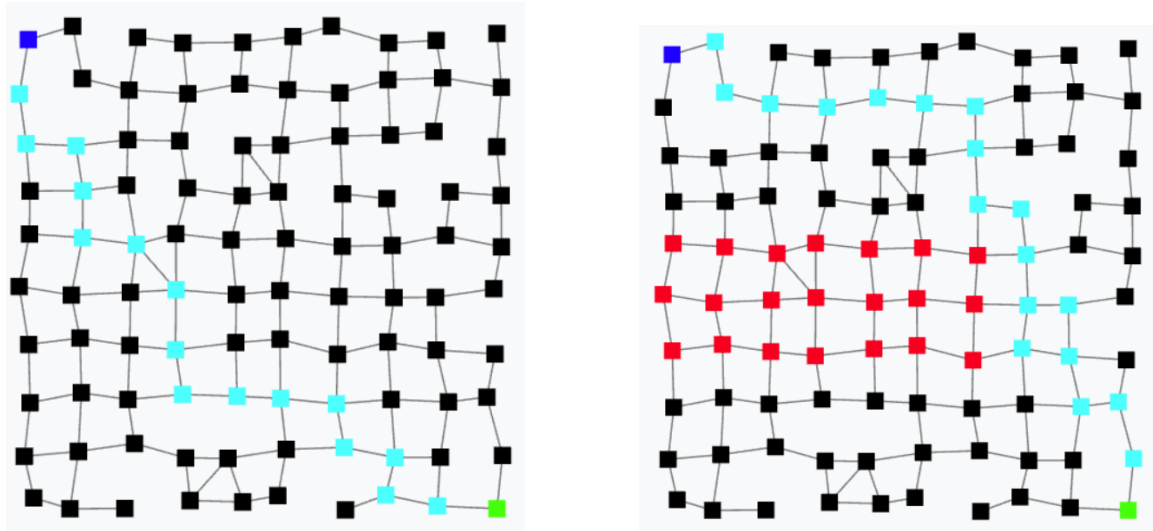
FIGURE 2. A pictorial view of the self-healing channel. The devices where the `channel` yields `true` are shown as cyan dots, whereas the blue and green dots denote the two endpoints of the channel, and red dots denote non-working devices (e.g., as a result of a blackout). Through the two snapshots, we see that a channel eventually self-adjust itself reacting to the perturbation induced by the blackout.

2.4. **Motivation.** Given the background on Aggregate Computing and field computations, we are now able to provide the motivation for this work. This is done in three parts: (i) we first motivate the usefulness of internal or embedded DSLs for field computations; then, (ii) we motivate the choice of Scala as our target host language; finally, (iii) we explain technical issues that have driven the design of NC in the attempt to embed field computations with a small set of requirements at the host side.

2.4.1. *Embedded DSLs for Field Computations.* A key issue for the practical applicability of the aggregate programming model enabled by field calculi is to find ways to smoothly integrate with the standard development practice—languages, processes and tools. We observe that an aggregate program is generally a small part of an overall application or system, addressing issues like advanced coordination and adaptivity, and interacting with other subsystems to gather inputs or provide system services [CV18].

Most former aggregate programming languages such as Proto [BB06] and Protelis [PVB15] are *standalone* DSLs. Also called *external* DSLs [Rit18], these languages have their own syntax and semantics, and are either interpreted or compiled to an intermediate language of some target platform (e.g., Java and the JVM, as in Protelis). Despite the flexibility and the support provided by modern language workbenches (cf. Xtext [Bet16]), effort in development and maintenance of an external DSL is typically large, high expertise is required, and, additionally, integration with other development languages and tools tends to be cumbersome.

A prominent, modern approach to address this problem is to devise an *internal DSL* [Voe13] (also called *embedded DSL* [Gho11]) that provides mechanisms to support

the new features on top of an adequate and well-known host programming language—through a very expressive API. This provides easy reuse of functionality and tools of the host language, for a smooth adoption. Given a source DSL and a target language, *DSL embedding* is the problem of implementing the DSL internally to the target language.

Field calculi deal with fields, i.e., collections of values, which are to be treated as first-class entities. However, common GPLs are subject to functional and typing constraints when dealing with "collections" so that point-wise manipulation typically requires widespread, explicit mapping operations which make the programs verbose. So far, this approach has been followed by the aggregate programming language FCPP [Aud20], a C++ internal DSL. The difficulties of implementing HFC as an internal DSL are partially addressed in FCPP by operator overloading, template metaprogramming techniques and macros; however, several low-level details are still left to the responsibility of the programmer.

2.4.2. *Scala as Host Language.* Both the syntax and the semantics of an embedded DSL are limited by the constraints exerted by its host language. Therefore, the choice of the host language is led by requirements and desiderata for the DSL. In our case these include:
- *pragmatism* (supporting easy reuse of existing programming structures and mechanisms),
- *reliability* (intercepting errors concerning syntax and types at compile-time), and
- *expressivity* (offering a concise and eloquent syntax, minimising the accidental language complexity induced by the environment);

which led us to select the Scala programming language. Indeed, Scala is a premier language for implementing embedded DSLs [AHKY15, Dub11, CDM+10, CRD+17]. It runs on the Java Virtual Machine (JVM) and thus enables straightforward interaction and reuse of libraries in the Java ecosystem. It has a flexible syntax, powerful type system, and convenient features (such as by-name parameters, type members, traits etc.) for building type-safe, expressive APIs. Additionally, its smooth integration of the object-oriented and functional paradigms [OR14], as well as its ecosystem of libraries for distributed computing, make it a convenient tool for in-large software development of frameworks. Functional programming support is especially important for implementing field calculi; however, compared with languages such as Haskell, it is more pragmatic and versatile. On the other hand, compared with very practical languages such as Kotlin, it is also (like Haskell) oriented towards programming language research and a philosophy of scalable abstractions [OZ05].

2.4.3. *Technical Issues in Embedding Field Computations in a Host Language.* A first key problem in embedding field computations in a host language lies in the potential mismatch between the representation of local values (values of local, standard types) and the representation of field expressions. For the former, one would seek for the natural representation of the host language, e.g., literal `1` representing a local value of type `Int`. For the latter, one has to combine the natural representation with the additional constructs manipulating fields; namely, as in field calculus, literal `1` (or some variation) should represent a computational field equal to 1 in all space-time points of the local device's neighbourhood: this should in principle be given a type `Field[Int]`, inheriting all the operations one would use over the local counterpart `Int` (`+`, `-`, and so on), since in field calculus `1+2` is the sum of two fields giving a field of 3 in all space-time points. As a consequence, standard types such as numbers, Booleans, characters, objects, and the corresponding operators, need to be coherently lifted in order to work under the field interpretation. That is, given any type `T`,

the type `Field[T]` should support all the operations provided by `T` but lifted to the field context: for instance, expression $e_1 + e_2$ where e_1 and e_2 have type `Field[Int]` should naturally give an element of type `Field[Int]`. Ideally, the type system should continue to do its job with field types, and field expressions should be written in a notation which is analogously simple as the local counterpart.

A second key problem stems from the declarativeness and compositionality of field computations: the host language should provide the means for defining blocks of code and for controlling *when* and *how* these are executed, mainly by deferring their evaluation until a later time and properly contextualising them to the local information available in each device. These mechanisms should also be lightweight so as to keep the impact on the user-side of the DSL as little as possible. For instance, using Java as host language would require many expression to be wrapped in a 0-ary lambda, which would clutter code.

The third, key technical difficulty is to properly deal with the interaction model of field computations, which is neighbour-driven. Field computations are equipped with a (logical or physical) notion of *neighbourhood* that basically expresses the boundary of direct communication, i.e., what devices can be reached by a given device through a communication act occurring in a certain position of computation. The observation of values that a device reads from its neighbours is typically modelled by reification into a *neighbouring value* (a map from neighbours to values), which can be manipulated functionally until being collapsed back to a local value by means of some *folding* operator—e.g., computing the minimum value. This requires one to explicitly differentiate, syntactically and semantically, the two classes of types, neighbouring types and local types, raising the issue of how to lift standard local operators to neighbour types. Thanks to the features of the Scala programming language, and as detailed in this paper, the problem could be handled as follows:

- the same type, say `Int`, is used both for local types and neighbouring types;
- the notion of computation over neighbouring values is semantically turned into a notion of computation "against" a neighbour (namely, a computation whose outcome depends on recent result of computation in that neighbour), hence there is no longer need of two kinds of type;
- folding operations are the triggers for a universal quantification process, iterating computations against all pertinent neighbours.

Such changes may impact theory and practice of field computations. On a practical level, they enable expressive and smooth integration with Scala programming. On a theoretical side, however, we need to investigate how these changes affect the expressiveness with respect to the field calculus [AVD+19].

3. The neighbours calculus

This section presents the *neighbours calculus (NC)*, a minimal core calculus that models Aggregate Computing via *computations against a neighbour*, instead of the neighbouring values used in the *higher-order field calculus (HFC)* [AVD+19]—a thoughtful comparison between NC and HFC is presented in Section 4.

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, so any (synchronous or

	Description		**Description**
P	Program	F	Function declaration
e	Expression	x	Variable
v	Value	ℓ	Local value
ϕ	Field value	f	Function value
d	Defined function	b	Built-in function
c	Data constructor	δ	Device identifier
T	Type	B	Built-in type
t	Type variable	TS	Type scheme
$\mathcal{D}$	Type-scheme environment	$\mathcal{B}$	Built-in type-scheme environment
$\mathcal{A}$	Type assumptions	θ	Value-tree
Θ	Value-tree environment	σ	(Local) Sensor state
Ψ	Value-tree field	α	Activation predicate
$Stat$	Status	Σ	(Global) Sensor state
Env	Environment	N	Network configuration
act	Action label		

TABLE 1. Summary of meta-variables and symbols used in this paper.

$$
\begin{array}{llr}
\text{P} & ::= \overline{\text{F}}\ \text{e} & \text{program} \\
\text{F} & ::= \texttt{def d}(\overline{\text{x}})\{\text{e}\} & \text{function declaration} \\
\text{e} & ::= \text{x} \mid \text{v} \mid (\overline{\text{x}})\ \texttt{=>}\ \{\text{e}\} \mid \text{e}(\overline{\text{e}}) \mid \texttt{rep(e)}\{\text{e}\} \mid \texttt{nbr}\{\text{e}\} & \text{expression} \\
& \mid\ \texttt{foldhood}(\text{e}, \text{e}, \text{e}) & \\
\text{v} & ::= \texttt{c}(\overline{\text{v}}) \mid \text{f} & \text{value} \\
\text{f} & ::= \text{b} \mid \text{d} \mid (\overline{\text{x}})\ \texttt{=>}\ \{\text{e}\} & \text{function value}
\end{array}
$$

FIGURE 3. Syntax of NC.

asynchronous) scheduling policy is admissible, even though in many aggregate programming settings it is typically assumed fair[2] and non-synchronous.

Section 3.1 presents the syntax of NC; Section 3.2 presents its type system; Section 3.3 presents an operational semantics for the computation that takes place on individual devices; and Section 3.4 presents an operational semantics for the evolution of whole networks.

3.1. **Syntax.** The syntax of NC is given in Figure 3. Following [IPW01], the overbar notation denotes metavariables over sequences and the empty sequence is denoted by •; e.g., for expressions, we let $\overline{\text{e}}$ range over sequences of expressions, written e_1, e_2, ... e_n ($n \geq 0$). NC focusses on aggregate programming constructs: hence, it is parametric in the set of built-in data constructors and functions. In the examples, we consider the set of built-in data constructors and functions listed (with their types) in Section 3.2. In following explanations, we use symbol δ (and variants as δ', δ_1, ...) to identify devices. For convenience, we also summarise meta-variables and symbols commonly used throughout the paper in Table 1.

A program P consists of a sequence $\overline{\text{F}}$ of function declarations and a main expression e. A function declaration F defines a (possibly recursive) function; it consists of a name d,

---

[2]An infinite scheduling sequence is *fair* if every device appears infinitely many times in the sequence.

$n \geq 0$ variable names $\overline{\mathtt{x}}$ representing the formal parameters, and an expression $\mathtt{e}$ representing the body of the function.

Expressions $\mathtt{e}$ are the main entities of the calculus, modelling a whole field computation. An expression can be: a variable $\mathtt{x}$, used as function formal parameter; a value $\mathtt{v}$; an anonymous function $(\overline{\mathtt{x}}) \Rightarrow \{\mathtt{e}\}$ (where $\overline{\mathtt{x}}$ are the formal parameters and $\mathtt{e}$ is the body), a function call $\mathtt{e}(\overline{\mathtt{e}})$; a $\mathtt{rep}$-expression $\mathtt{rep}(\mathtt{e})\{\mathtt{e}\}$, modelling time evolution; an $\mathtt{nbr}$-expression $\mathtt{nbr}\{\mathtt{e}\}$, modelling neighbourhood interaction; or a $\mathtt{foldhood}$-expression $\mathtt{foldhood}(\mathtt{e}, \mathtt{e}, \mathtt{e})$ which combines values obtained from neighbours.

The set of the *free variables* of an expression $\mathtt{e}$, denoted by $\mathbf{FV}(\mathtt{e})$, is defined as usual (the only binding construct is $(\overline{\mathtt{x}}) \Rightarrow \{\mathtt{e}\}$). An expression $\mathtt{e}$ is *closed* if $\mathbf{FV}(\mathtt{e}) = \bullet$. The main expression of a program must be closed, whereas anonymous functions $(\overline{\mathtt{x}}) \Rightarrow \{\mathtt{e}\}$ may be *open* (i.e., not closed).

A value can be either a *data value* $\mathtt{c}(\overline{\mathtt{v}})$ or a *functional value* $\mathtt{f}$. A data value consists of a *data constructor* $\mathtt{c}$ of some arity $m \geq 0$ applied to a sequence of m data values $\overline{\mathtt{v}} = \mathtt{v}_1, ..., \mathtt{v}_m$. For readability, the parenthesis may be omitted for arity $m = 0$, writing $\mathtt{c}()$ as $\mathtt{c}$. According to the data constructors listed in Figure 5, examples of data values are: the Booleans $\mathtt{True}$ and $\mathtt{False}$, numbers, pairs (like $\mathtt{Pair}(\mathtt{True}, \mathtt{Pair}(5, 7))$) and lists (like $\mathtt{Cons}(3, \mathtt{Cons}(4, \mathtt{Null}))$).

Functional values $\mathtt{f}$ comprise:

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

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

The key constructs of the calculus are:

- *Function call:* $\mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ is the main construct of the language. The function call evaluates to the result of applying the function value $\mathtt{f}$ produced by the evaluation of $\mathtt{e}$ to the value of the parameters $\mathtt{e}_1, \ldots, \mathtt{e}_n$ *relatively to the aligned neighbours*, that is, relatively to the neighbours that in their last execution round have evaluated $\mathtt{e}$ to a function value with the same name of $\mathtt{f}$. Hence, calling an expression $\mathtt{e}$ of function type acts as a branch, where each function $\mathtt{f}$ obtainable from $\mathtt{e}$ is applied only on the subspace of devices which evaluated $\mathtt{e}$ to a (syntactically) identical function $\mathtt{f}$.
- *Time evolution:* $\mathtt{rep}(\mathtt{e}_1)\{\mathtt{e}_2\}$ is a construct for dynamically changing fields through the "$\mathtt{rep}$eated" application of the functional expression $\mathtt{e}_2$. 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), $\mathtt{e}_2$ is applied to $\mathtt{e}_1$, then at each other step it is applied to the value obtained at the previous step. For instance, $\mathtt{rep}(0)\{(\mathtt{x}) \Rightarrow \{\mathtt{x} + 1\}\}$ counts how many rounds each device has computed (from the beginning, or more generally, since that piece of state was missing).

- *Neighbourhood interaction:* $\mathtt{foldhood}(e_1, e_2, e_3)$ and $\mathtt{nbr\{e\}}$ model device-to-device inter-
action, and are at the core of the "computation against a neighbour" mechanism. The
$\mathtt{foldhood}$ construct evaluates expression e_3 against every aligned neighbour (excluding
the device itself), then aggregates the values collected through binary operator e_2 together
with the initial value e_1. The interaction pattern is thus a common "map (expression e_3 to
each neighbours' data) and reduce (folding through e_2)". Following standard practice, data
aggregation is performed through binary folding, which smoothly covers most application
scenarios.[3] Used inside e_3, the $\mathtt{nbr}$ construct tags a sub-expression e signalling that, when
evaluated against a neighbour δ, it should not be actually evaluated as usual, but should
give as result the one obtained by evaluating e in δ. Put in other words, when evaluated
against δ, $\mathtt{nbr\{e\}}$ means "observing" the recent resulting value of e in δ—and also let
later δ observe the local valued of e, conversely. Such behaviour requires to effectively
broadcast[4] the values evaluated for e, in order to make them accessible to neighbour
devices. Subexpressions of e_3 not containing $\mathtt{nbr}$ are evaluated as usual instead, i.e., with
no gathering of information from neighbours.

  As an example, consider the expression

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

  evaluated in device δ_1 (in which $\mathtt{temperature}() = 10$) with neighbours δ_2 and δ_3 (in which
$\mathtt{temperature}()$ gave 15 and 5 in their last evaluation round, orderly). The result of the
expression is then computed adding 2, $\mathtt{min}(15, 10)$ and $\mathtt{min}(5, 10)$ for a final value of 17.

Additional syntactic sugar. To facilitate specification of examples, we write $\mathtt{if}(e_1)\{e_2\}\{e_3\}$
as a shorthand for $\mathtt{mux}(e_1, () \Rightarrow \{e_2\}, () \Rightarrow \{e_3\})()$, where the multiplexer function $\mathtt{mux}$
selects between its second and third argument based on the value of the first, in formulas:
$\mathtt{mux}(b, v_\top, v_\bot) = v_b$. Additionally, in order to improve readability, we also sometimes write
$\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ as syntactic sugar for $((x) \Rightarrow \{e_2\})(e_1)$.

3.2. **Typing.** We now present a type system for NC. Since the type system is a customisation
of the Hindley-Milner type system [DM82], there is an algorithm (not presented here) that,
given an expression e and type assumptions for its free variables, either fails (if the expression
cannot be typed under the given type assumptions) or returns its *principal type*, i.e., a type
such that all the types that can be assigned to e by the type inference rules can be obtained
from the principal type by substituting type variables with types. The syntax of type and
type schemes is presented in Figure 4 (top), where B ranges over the built-in types provided
by the host language (e.g., $\mathtt{num}$, $\mathtt{bool}$, $\mathtt{pair}(T_1, T_2)$ for any T_1 and T_2, $\mathtt{list}(T)$ for any T).
We do not expand on the structure of built-in types and type operators, as it is not needed
to define NC typing. So, a type T is either a built-in type B, a function type $(\overline{T}) \to T$,
or a type variable t. A type scheme has syntax $\forall \bar{t}. T$, and denotes a type quantifying over
type variables. The set of type variables occurring in a type T is denoted by $\mathbf{FTV}(T)$.
Type variables $\bar{t}$ occurring in a type T can be substituted with concrete types $\overline{T}$ through
the common bracket-notation $T[\bar{t} := \overline{T}]$.

---

[3]In case where a non-binary aggregation is needed, it is still possible to fold neighbour's data into a list
through a binary join operator, and then process the resulting list with the needed aggregation.

[4]Even though $\mathtt{nbr}$ requires to share values with neighbours, we do not assume the underlying implementation
to be necessarily based on network broadcasts. Indeed, messages from different $\mathtt{nbr}$s are usually packed
together, and may be dispatched to neighbours even through different point-to-point messages.

Types:
$$T \quad ::= \quad t \mid B \mid (\overline{T}) \to T \qquad\qquad \text{type}$$
$$TS \quad ::= \quad \forall \overline{t}.T \qquad\qquad \text{type scheme}$$

Expression typing: $\boxed{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T}$

$$[\text{T-VAR}] \quad \frac{}{\mathcal{D}; \mathcal{A}, \texttt{x} : T \vdash \texttt{x} : T}$$

$$[\text{T-DAT}] \quad \frac{T'[\overline{t} := \overline{T}''] = (\overline{T}) \to T \qquad \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{v}} : \overline{T}}{\mathcal{D}, \texttt{c} : \forall \overline{t}.T'; \mathcal{A} \vdash \texttt{c}(\overline{\texttt{v}}) : T}$$

$$[\text{T-A-FUN}] \quad \frac{\mathcal{D}; \mathcal{A}, \overline{\texttt{x}} : \overline{T} \vdash \texttt{e} : T}{\mathcal{D}; \mathcal{A} \vdash (\overline{\texttt{x}}) \Rightarrow \{\texttt{e}\} : (\overline{T}) \to T}$$

$$[\text{T-N-FUN}] \quad \frac{\texttt{f} \text{ is a (built-in or declared) function}}{\mathcal{D}, \texttt{f} : \forall \overline{t}.T; \mathcal{A} \vdash \texttt{f} : T[\overline{t} := \overline{T}]}$$

$$[\text{T-APP}] \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : (\overline{T}) \to T \qquad \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{e}} : \overline{T}}{\mathcal{D}; \mathcal{A} \vdash \texttt{e}(\overline{\texttt{e}}) : T}$$

$$[\text{T-REP}] \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : T \qquad \mathcal{D}; \mathcal{A} \vdash \texttt{e}_2 : (T) \to T}{\mathcal{D}; \mathcal{A} \vdash \texttt{rep}(\texttt{e}_1)\{\texttt{e}_2\} : T}$$

$$[\text{T-NBR}] \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T}{\mathcal{D}; \mathcal{A} \vdash \texttt{nbr}\{\texttt{e}\} : T}$$

$$[\text{T-FOLD}] \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : T \quad \mathcal{D}; \mathcal{A} \vdash \texttt{e}_2 : (T, T) \to T \quad \mathcal{D}; \mathcal{A} \vdash \texttt{e}_3 : T}{\mathcal{D}; \mathcal{A} \vdash \texttt{foldhood}(\texttt{e}_1, \texttt{e}_2, \texttt{e}_3) : T}$$

Function typing: $\boxed{\mathcal{D} \vdash \texttt{F} : TS}$

$$[\text{T-FUNCTION}] \quad \frac{\mathcal{D}, \texttt{d} : \forall \bullet.(\overline{T}) \to T; \overline{\texttt{x}} : \overline{T} \vdash \texttt{e} : T \qquad \overline{t} = \mathbf{FTV}((\overline{T}) \to T)}{\mathcal{D} \vdash \texttt{def } \texttt{d}(\overline{\texttt{x}})\{\texttt{e}\} : \forall \overline{t}.(\overline{T}) \to T}$$

Program typing: $\boxed{\vdash \texttt{P} : T}$

$$[\text{T-PROGRAM}]$$
$$\frac{\begin{array}{l} \mathcal{D}_0 = \mathcal{B} \\ \texttt{F}_i = \texttt{def } \texttt{d}_i(\_)\{\_\} \qquad \mathcal{D}_{i-1} \vdash \texttt{F}_i : TS_i \qquad \mathcal{D}_i = \mathcal{D}_{i-1}, \texttt{d}_i : TS_i \qquad (i \in 1..n) \\ \mathcal{D}_n; \emptyset \vdash \texttt{e} : T \end{array}}{\vdash \texttt{F}_1 \cdots \texttt{F}_n \texttt{ e} : T}$$

FIGURE 4. Type rules for expressions, function declarations, and programs.

Type environments, ranged over by $\mathcal{A}$ and written $\overline{\texttt{x}} : \overline{T}$, are used to collect type assumptions for program variables (i.e., formal parameters of functions). *Type-scheme environments*, ranged over by $\mathcal{D}$ and written $\overline{\texttt{v}} : \overline{TS}$, are used to collect the type schemes for built-in constructors and built-in operators together with the type schemes inferred for user-defined functions. In particular, the distinguished *built-in type-scheme environment* $\mathcal{B}$ associates a type scheme to each built-in constructor $\texttt{c}$ and to each built-in function $\texttt{b}$—Figure 5 shows the type schemes for the built-in constructors and built-in functions used throughout this paper.

The typing judgement for expressions is of the form "$\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T$", to be read: "$\texttt{e}$ has type T under the type-scheme assumptions $\mathcal{D}$ (for built-in constructors and for built-in and user-defined functions) and the type assumptions $\mathcal{A}$ (for the program variables occurring in $\texttt{e}$), respectively". As a standard syntax in type systems [IPW01], given $\overline{T} = T_1, \ldots, T_n$ and $\overline{\texttt{e}} = \texttt{e}_1, \ldots, \texttt{e}_n$ ($n \geq 0$), we write $\mathcal{D}; \mathcal{A} \vdash \overline{\texttt{e}} : \overline{T}$ as short for $\mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : T_1 \cdots \mathcal{D}; \mathcal{A} \vdash \texttt{e}_n : T_n$.

The typing rules for expressions are presented in Figure 4 (bottom). The rules for variables ([T-VAR]), data values ([T-DAT]), anonymous function expressions ([T-A-FUN]), built-in or defined function names ([T-N-FUN]), and function application ([T-APP]), are almost standard. Rule [T-REP] (for $\texttt{rep}$-expressions) ensures that both the initial value $\texttt{e}_1$ and the domain

Built-in data constructors

$$\mathcal{B}(\texttt{True}) \quad = \quad \texttt{bool}$$
$$\mathcal{B}(\texttt{False}) \quad = \quad \texttt{bool}$$
$$\mathcal{B}(n) \qquad\quad = \quad \texttt{num}, \quad \text{where } n \text{ is a number or } \texttt{PositiveInfinity}$$
$$\mathcal{B}(\texttt{Pair}) \quad = \quad \forall t_1 t_2.(t_1, t_2) \to \texttt{pair}(t_1, t_2)$$
$$\mathcal{B}(\texttt{Null}) \quad = \quad \forall t.\texttt{list}(t)$$
$$\mathcal{B}(\texttt{Cons}) \quad = \quad \forall t.(t, \texttt{list}(t)) \to \texttt{list}(t)$$

Built-in functions: pure operators

$$\mathcal{B}(\texttt{pair}) \quad = \quad \forall t_1 t_2.t_1 \to t_2 \to \texttt{pair}(t_1, t_2)$$
$$\mathcal{B}(\texttt{fst}) \quad = \quad \forall t_1 t_2.(\texttt{pair}(t_1, t_2)) \to t_1$$
$$\mathcal{B}(\texttt{snd}) \quad = \quad \forall t_1 t_2.(\texttt{pair}(t_1, t_2)) \to t_2$$
$$\mathcal{B}(\texttt{cons}) \quad = \quad \forall t.t \to \texttt{list}(t) \to \texttt{list}(t)$$
$$\mathcal{B}(\texttt{head}) \quad = \quad \forall t.(\texttt{list}(t)) \to t$$
$$\mathcal{B}(\texttt{tail}) \quad = \quad \forall t.(\texttt{list}(t)) \to \texttt{list}(t)$$
$$\mathcal{B}(\texttt{=}) \qquad = \quad \forall t.(t, t) \to \texttt{bool}$$
$$\mathcal{B}(\texttt{mux}) \quad = \quad \forall t.(\texttt{bool}, t, t) \to t$$
$$\mathcal{B}(\texttt{+}) \qquad = \quad (\texttt{num}, \texttt{num}) \to \texttt{num}$$
$$\mathcal{B}(\texttt{and}) \quad = \quad (\texttt{bool}, \texttt{bool}) \to \texttt{bool}$$
$$\mathcal{B}(\texttt{min}) \quad = \quad (\texttt{num}, \texttt{num}) \to \texttt{num}$$
$$\mathcal{B}(\texttt{<}) \qquad = \quad (\texttt{num}, \texttt{num}) \to \texttt{bool}$$

Built-in functions: sensors

$$\mathcal{B}(\texttt{temperature}) \quad = \quad () \to \texttt{num}$$

Built-in functions: relational sensors

$$\mathcal{B}(\texttt{nbrRange}) \quad = \quad () \to \texttt{num}$$

Figure 5. Type schemes for the built-in value constructors and functions used in the examples.

and range of function e_2 have the same type, and then assigns it to $\texttt{rep}(e_1)\{e_2\}$; rule [T-NBR] (for $\texttt{nbr}$-expressions) assigns to $\texttt{nbr}\{e\}$ the same type as e; and rule [T-FOLD] (for $\texttt{foldhood}$-expressions) ensures that e_1 and e_3 have the same type T and that e_2 has type $(T, T) \to T$, and then assigns type T to $\texttt{foldhood}(e_1, e_2, e_3)$. The typing rules for declared functions ([T-FUNCTION]) and programs ([T-PROGRAM]) are almost standard and leverage judgements of the form $\mathcal{D} \vdash \texttt{F} : \mathit{TS}$ and $\mathcal{D}_0 \vdash \texttt{P} : T$, respectively for a function $\texttt{F}$ and program $\texttt{P}$.

Example 3.1 (Typing). Consider the following simple implementation of a self-healing gradient [ACDV17] (i.e., the field of minimum distances from source devices) that also circumvents obstacles. Its semantics will be presented later (see Examples 3.10 and 3.11); now, we merely consider its typing.

```
def gradient(source, metric) { // : (bool, ()->num) -> num
  rep(PositiveInfinity){ (distance) => {
    mux(source, 0.0,
      foldhood(PositiveInfinity, min, nbr{distance} + metric())
    )
  }}
}
```

```
if (isObstacle) { PositiveInfinity } { gradient(isSource) }  // : num
```

The types of the gradient function and of the main expression inferred by the type system are inserted above as comments. By rule [T-APP] and assumptions on built-in mux, the type system infers that the third argument of the mux expression must be num, since the second argument is also num. It follows that distance must be of type num (rule [T-NBR]) as well and metric must be of type () → num (rule [T-APP]), from which function gradient can be inferred to have type (bool, () → num) → num (rule [T-FUNCTION]). The overall program has then type num (rule [T-PROGRAM]).

3.3. Operational Semantics: Device Semantics.

This section presents a formal semantics of device computation as happens in NC. Starting from NC syntax as previously described, we assume a fixed program P. We say that "device δ *fires*", to mean that the main expression of P is evaluated on δ.

Remark 3.2 (On termination of device computation). As NC allows recursive functions, termination of a device firing is not decidable. In the rest of the paper we assume that only terminating programs are considered.
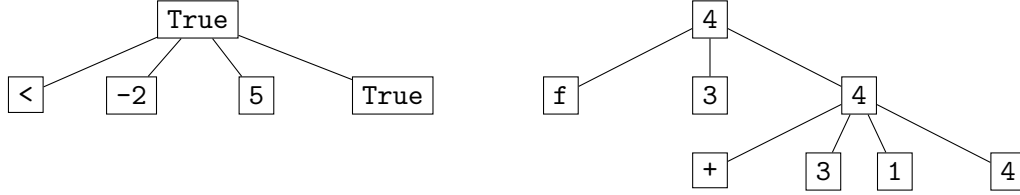
3.3.1. *Device semantics: overall picture and preliminary definitions.* We model device computation by a big-step operational semantics where the result of evaluation is a *value-tree* θ (see Figure 6, first frame), which is an ordered tree of values, tracking the result of any evaluated subexpression. Intuitively, the evaluation of an expression at a given time in a device δ is performed against the recently-received value-trees of neighbours, namely, its outcome depends on those value-trees. The result is a new value-tree that is conversely made available to δ's neighbours (through a broadcast) for their firing; this includes δ itself, so as to support a form of state across computation rounds (note that an implementation may massively compress the value-trees, storing only enough information for expressions to be aligned). In other words, a value-tree is a data structure that embeds all of the following: (i) the final and partial results of a local computation; (ii) state resulting from a previous computation; (iii) incoming data from neighbours; and (iv) outcoming data to be sent to neighbours. Trees are in general a suitable representation for denoting programs (statically) and computations (dynamically); notice that value-trees may differ in structure if different sub-computations are chosen e.g. due to branching (cf. if). In particular, state and communication data are located at specific nodes – corresponding to rep and nbr construct applications – which are the only ones that may be actually stored and transmitted, but whose position is important for the evaluation semantics.

A *value-tree environment* Θ is a map from device identifiers to value-trees, a local view of the outcomes of the last evaluations on neighbours, as it is locally reconstructed through received messages (no global knowledge is ever assumed). This is written $\bar{\delta} \mapsto \bar{\theta}$ as short for $\delta_1 \mapsto \theta_1, \ldots, \delta_n \mapsto \theta_n$. The syntax of value-trees and value-tree environments is given in Figure 6 (first frame).

Example 3.3 (Value-trees). The graphical representation of the value-trees

$$\theta_1 = \texttt{True}\langle < \langle \rangle, -2\langle \rangle, 5\langle \rangle, \texttt{True}\langle \rangle \rangle \qquad \text{and} \qquad \theta_2 = 4\langle f\langle \rangle, 3\langle \rangle, 4\langle +\langle \rangle, 3\langle \rangle, 1\langle \rangle, 4\langle \rangle \rangle \rangle$$

is as follows:

In the following, for sake of readability, we sometimes write the value v as shorthand for the value-tree $v\langle\rangle$. Following this convention, the value-tree θ_1 is shortened to $\texttt{True}\langle <, -2, 5, \texttt{True}\rangle$, and the value-tree θ_2 is shortened to $4\langle\texttt{f}, 3, 4\langle+, 3, 1, 4\rangle\rangle$.

We assume that prior to execution each anonymous function sub-expression $(\bar{\texttt{x}})\texttt{=> \{e\}}$ of a program $\texttt{e}_{\texttt{main}}$ is automatically annotated as $(\bar{\texttt{x}}) \overset{\tau}{\texttt{=>}} \texttt{\{e\}}$ with a tag τ which uniquely identifies the expression.[5] The tag serves as a *name* for anonymous functions, allowing to consider function values to be equal if they share the same name. This notion of syntactic equality will later be exploited for the definition of the semantics of function calls.

Remark 3.4 (Function Equality). Notice that *behavioural equivalence* of function expressions is not decidable in a Turing-complete language, and thus cannot be used in a formal semantics of a (practically executable) programming language. Pure syntactic equality could instead be used, but it would fail to correctly relate multiple occurrences of an anonymous function capturing a variable, whenever that gets substituted for different values. The proposed naming system prevents those issues, while also avoiding unintended interferences between functions that happen to be syntactically identical, but were written in different units of a program and were not meant to be integrated with each other.

Figure 6 (second frame) defines: the auxiliary functions ρ and π for extracting the root value and a subtree of a value-tree, respectively (further explanations about function π will be given later); the extension of functions ρ and π to value-tree environments; and the auxiliary functions *name*, *args* and *body* for extracting the name, formal parameters and body of a (user-defined or anonymous) function, respectively.

Example 3.5 (Auxiliary functions on value-trees). Consider the following value-trees.

$$\theta_1 = \texttt{True}\langle < \langle\rangle, -2\langle\rangle, 5\langle\rangle, \texttt{True}\langle\rangle\rangle \qquad \text{and} \qquad \theta_2 = 4\langle f\langle\rangle, 3\langle\rangle, 4\langle+\langle\rangle, 3\langle\rangle, 1\langle\rangle, 4\langle\rangle\rangle\rangle$$

The following picture graphically shows the nodes denoted by auxiliary functions working on individual value trees.
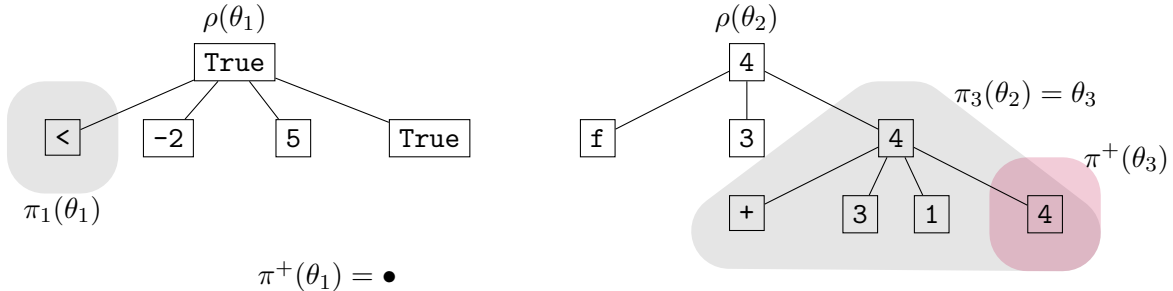
The computation that takes place on a single device is formalised by the big-step operational semantics rules given in Figure 6 (fourth frame). The derived judgements are of

---

[5] For example, the tag could be generated as $\tau = (\texttt{e}_{\texttt{main}}, n)$ where n is the index of the occurrence of the => keyword in $\texttt{e}_{\texttt{main}}$.

the form

$$\delta, \delta'; \Theta; \sigma \vdash \mathsf{e} \Downarrow \theta$$

to be read "expression e evaluates to value-tree θ on device δ with respect to the neighbour δ', value-tree environment Θ and sensor state σ", where: *(i)* δ is the identifier of the current device and δ' is either equal to δ or is one of its neighbours; *(ii)* Θ is the field of the value-trees produced by the most recent evaluation of (an expression corresponding to) e on δ and its neighbours, which were received by δ through messages; *(iii)* e is an expression; *(iv)* the value-tree θ represents the values computed for all the expressions encountered during the evaluation of e—in particular $\rho(\theta)$ is the result value of e.

The operational semantics rules are based on rather standard rules for functional languages, extended so as to be able to evaluate a subexpression e' of e with respect to the value-tree environment Θ' obtained from Θ by extracting the corresponding subtree (when present) in the value-trees in the range of Θ. This process, called *alignment*, is modelled by the auxiliary function π, defined in Figure 6 (second frame). Function π has two different behaviours (specified by its subscript or superscript): $\pi_i(\theta)$ extracts the i-th subtree of θ, if it is present; and $\pi^{\mathsf{f}}(\theta)$ extracts the last subtree of θ, if it is present and the root of first subtree of θ is equal to f.

When a device δ fires, its main expression e is evaluated with respect to δ itself. That is, by means of a judgement where $\delta' = \delta$:

$$\delta, \delta; \Theta; \sigma \vdash \mathsf{e} \Downarrow \theta.$$

A key aspect of the semantics is that, if e is a $\mathtt{foldhood}$-expression $\mathtt{foldhood}(\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3)$ then its body e_3 is evaluated with respect to each of the devices δ' (if any) in $\mathbf{dom}(\Theta) \setminus \{\delta\}$. Because of alignment (see above), it might happen that a sub-expression e' of e_3 is evaluated by a judgement

$$\delta, \delta'; \Theta; \sigma \vdash \mathsf{e}' \Downarrow \theta \quad \text{where } \delta \neq \delta' \notin \mathbf{dom}(\Theta)$$

and, if the evaluation of e' exploits the device δ', then the evaluation of e_3 with respect to δ' *fails* and the evaluation of the $\mathtt{foldhood}$-expression $\mathtt{foldhood}(\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3)$ does not consider the neighbour δ'. The evaluation rule for $\mathtt{foldhood}$-expressions, [E-FOLD], formalises failure of evaluation with respect to a neighbour δ' by means of the auxiliary predicate

$$\delta, \delta'; \Theta; \sigma \vdash \mathsf{e} \text{ FAIL}$$

to be read "expression e fails to evaluate on device δ against neighbour δ' with respect to value-tree environment Θ and sensor state σ", which is formalised by the big-step operational semantics rules given in Figure 7.

3.3.2. *Device semantics: rules for expression evaluation.* We start by explaining the rules in Figure 6 (fourth frame), then we will explain the rules in Figure 7.

Rule [E-VAL] implements the evaluation of an expression that is already a value. For instance, evaluating the expression $\mathtt{1}$ produces (by Rule [E-VAL]) the value-tree $\mathtt{1}\langle\rangle$, while evaluating the expression $+$ produces the value-tree $+\langle\rangle$.

Rules [E-B-APP] and [E-D-APP] model function application $\mathsf{e}(\mathsf{e}_1 \cdots \mathsf{e}_n)$. In case e evaluates to a built-in function b, rule [E-B-APP] is used, whose behaviour is driven by the special auxiliary function $(\!|\mathsf{b}|\!)_{\delta, \delta'}^{\Theta, \sigma}$ (operational interpretation of b), whose actual definition is abstracted away.

Value-trees and value-tree environments:

$$\theta \quad ::= \quad \mathtt{v}\langle \overline{\theta}\rangle \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{value-tree}$$
$$\Theta \quad ::= \quad \overline{\delta} \mapsto \overline{\theta} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{value-tree environment}$$

Auxiliary functions:

$$name((\overline{\mathtt{x}}) \overset{\tau}{\Rightarrow} \{\mathtt{e}\}) = \tau \qquad args((\overline{\mathtt{x}}) \overset{\tau}{\Rightarrow} \{\mathtt{e}\}) = \overline{\mathtt{x}} \qquad body((\overline{\mathtt{x}}) \overset{\tau}{\Rightarrow} \{\mathtt{e}\}) = \mathtt{e}$$
$$name(\mathtt{d}) = \mathtt{d} \qquad\qquad\quad args(\mathtt{d}) = \overline{\mathtt{x}} \qquad\qquad\quad body(\mathtt{d}) = \mathtt{e} \ (\text{if } \mathtt{def\ d}(\overline{\mathtt{x}})\{\mathtt{e}\})$$
$$name(\mathtt{b}) = \mathtt{b} \qquad\qquad\quad \rho(\mathtt{v}\langle\overline{\theta}\rangle) = \mathtt{v}$$
$$\pi_i(\mathtt{v}\langle\theta_1,\ldots,\theta_n\rangle) = \theta_i \qquad \text{if } 1 \le i \le n \qquad\qquad\qquad \text{else } \bullet$$
$$\pi^{\mathtt{f}}(\mathtt{v}\langle\theta_1,\ldots,\theta_{n+2}\rangle) = \theta_{n+2} \ \text{ if } name(\rho(\theta_1)) = name(\mathtt{f}) \quad \text{else } \bullet$$

$$\text{For } aux \in \rho, \pi_i, \pi^{\mathtt{f}}: \quad \left\{ \begin{array}{lll} aux(\bullet) & = & \bullet \\ aux(\delta \mapsto \theta, \Theta) & = & aux(\Theta) \qquad\qquad\qquad \text{if } aux(\theta) = \bullet \\ aux(\delta \mapsto \theta, \Theta) & = & \delta \mapsto aux(\theta), aux(\Theta) \quad \text{if } aux(\theta) \neq \bullet \end{array} \right.$$

Syntactic shorthands:

$$\delta, \delta'; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta} \ \text{ where } |\overline{\mathtt{e}}| = n \ \text{for} \ \delta, \delta'; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 \ldots \delta, \delta'; \pi_n(\Theta); \sigma \vdash \mathtt{e}_n \Downarrow \theta_n$$
$$\rho(\overline{\theta}) \qquad\qquad\qquad \text{where } |\overline{\theta}| = n \ \text{for} \ \rho(\theta_1) \ \ldots \ \rho(\theta_n)$$
$$\overline{\mathtt{x}} := \rho(\overline{\theta}) \qquad\qquad \text{where } |\overline{\mathtt{x}}| = n \ \text{for} \ \mathtt{x}_1 := \rho(\theta_1) \ \ldots \ \mathtt{x}_n := \rho(\theta_n)$$

Rules for expression evaluation: $\boxed{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e} \Downarrow \theta}$

$$\frac{}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{v} \Downarrow \mathtt{v}\langle\rangle} \text{[E-VAL]}$$

$$\text{[E-B-APP]} \ \frac{\begin{array}{c} \delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta \quad \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \mathtt{e}_i \Downarrow \theta_i \quad \text{for all } i \in 1,\ldots,n \\ \mathtt{v} = (\!|\mathtt{b}|\!)_{\delta,\delta'}^{\pi^{\mathtt{b}}(\Theta),\sigma}(\rho(\overline{\theta})) \quad (\mathtt{b} = \rho(\theta) \text{ is not relational}) \vee (\delta' \in \mathbf{dom}(\pi^{\mathtt{b}}(\Theta)) \cup \{\delta\}) \end{array}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \Downarrow \mathtt{v}\langle\theta, \overline{\theta}, \mathtt{v}\rangle}$$

$$\text{[E-D-APP]} \ \frac{\begin{array}{c} \delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta \qquad \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \mathtt{e}_i \Downarrow \theta_i \quad \text{for all } i \in 1,\ldots,n \\ \mathtt{f} = \rho(\theta) \text{ is not a built-in} \quad \delta, \delta'; \pi^{\mathtt{f}}(\Theta); \sigma \vdash body(\mathtt{f})[args(\mathtt{f}) := \rho(\overline{\theta})] \Downarrow \theta' \end{array}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \Downarrow \rho(\theta')\langle\theta, \overline{\theta}, \theta'\rangle}$$

$$\text{[E-REP]} \ \frac{\begin{array}{cc} \delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 & \mathtt{v}_1 = \rho(\theta_1) \\ \delta, \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2(\mathtt{v}_0) \Downarrow \theta_2 & \mathtt{v}_2 = \rho(\theta_2) \end{array} \quad \mathtt{v}_0 = \left\{ \begin{array}{ll} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathbf{dom}(\Theta) \\ \mathtt{v}_1 & \text{otherwise} \end{array} \right.}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{rep}(\mathtt{e}_1)\{\mathtt{e}_2\} \Downarrow \mathtt{v}_2\langle\theta_1, \theta_2\rangle}$$

$$\text{[E-NBR]} \ \frac{\delta \neq \delta' \in \mathbf{dom}(\Theta) \qquad \theta = \Theta(\delta')}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{nbr}\{\mathtt{e}\} \Downarrow \theta} \qquad \text{[E-NBR-LOC]} \ \frac{\delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta}{\delta, \delta; \Theta; \sigma \vdash \mathtt{nbr}\{\mathtt{e}\} \Downarrow \rho(\theta)\langle\theta\rangle}$$

$$\text{[E-FOLD]} \ \frac{\begin{array}{ll} \delta, \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta^1 & \delta, \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2 \Downarrow \theta_f \quad \mathtt{f} = \rho(\theta_f) \\ \delta_1, \ldots, \delta_n = \mathbf{dom}(\Theta) \setminus \{\delta\} & n \ge m \ge 0, \delta_1, \ldots, \delta_m \text{ increasing}, \delta_0 = \delta \\ \delta, \delta_i; \pi_3(\Theta); \sigma \vdash \mathtt{e}_3 \Downarrow \theta_i & \text{for all } i \in 0, \ldots, m \\ \delta, \delta_j; \pi_3(\Theta); \sigma \vdash \mathtt{e}_3 \ \text{FAIL} & \text{for all } j \in m+1, \ldots, n \\ \delta, \delta; \emptyset; \sigma \vdash \mathtt{f}(\rho(\theta^i), \rho(\theta_i)) \Downarrow \theta^{i+1} & \text{for all } i \in 1, \ldots, m \end{array}}{\delta, \delta'; \Theta; \sigma \vdash \mathtt{foldhood}(\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3) \Downarrow \rho(\theta^{m+1})\langle\theta^1, \theta_f, \theta_0\rangle}$$

Figure 6. Big-step operational semantics for expression evaluation (see also Figure 7).

$$\boxed{\textbf{Auxiliary rules for expression evaluation failure:} \qquad \delta, \delta'; \Theta; \sigma \vdash \texttt{e} \ \text{FAIL}}$$

$$[\text{E-NBR-FAIL}] \quad \frac{\delta \neq \delta' \notin \mathbf{dom}(\Theta)}{\delta, \delta'; \Theta; \sigma \vdash \texttt{nbr\{e\}} \ \text{FAIL}}$$

$$[\text{E-R-APP-FAIL}] \quad \frac{\begin{array}{ll} \delta, \delta; \pi_1(\Theta); \sigma \vdash \texttt{e} \Downarrow \theta & \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \texttt{e}_i \Downarrow \theta_i \text{ for all } i \in 1, \ldots, n \\ \texttt{r} = \rho(\theta) \text{ is relational-built-in} & \delta \neq \delta' \notin \mathbf{dom}(\pi^{\texttt{r}}(\Theta)) \end{array}}{\delta, \delta'; \Theta; \sigma \vdash \texttt{e}(\overline{\texttt{e}}) \ \text{FAIL}}$$

$$[\text{E-APP-ARG-FAIL}] \quad \frac{\begin{array}{ll} \delta, \delta; \pi_1(\Theta); \sigma \vdash \texttt{e} \Downarrow \theta & \overline{\texttt{e}} = \texttt{e}_1, \ldots, \texttt{e}_n \qquad n \geq 0 \\ \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \texttt{e}_i \Downarrow \theta_i & \text{for all } i \in 1, \ldots, m < n \\ \delta, \delta'; \pi_{m+2}(\Theta); \sigma \vdash \texttt{e}_{m+1} \ \text{FAIL} \end{array}}{\delta, \delta'; \Theta; \sigma \vdash \texttt{e}(\overline{\texttt{e}}) \ \text{FAIL}}$$

$$[\text{E-D-APP-FAIL}] \quad \frac{\begin{array}{ll} \delta, \delta; \pi_1(\Theta); \sigma \vdash \texttt{e} \Downarrow \theta & \delta, \delta'; \pi_{i+1}(\Theta); \sigma \vdash \texttt{e}_i \Downarrow \theta_i \text{ for all } i \in 1, \ldots, n \\ \texttt{f} = \rho(\theta) \text{ is not a built-in} & \delta, \delta'; \pi^{\texttt{f}}(\Theta); \sigma \vdash body(\texttt{f})[args(\texttt{f}) := \rho(\overline{\theta})] \ \text{FAIL} \end{array}}{\delta, \delta'; \Theta; \sigma \vdash \texttt{e}(\overline{\texttt{e}}) \ \text{FAIL}}$$

FIGURE 7. Big-step operational semantics for expression evaluation (auxiliary rules for expression evaluation failure).

Example 3.6 (Built-in function application). Evaluating the expression $< (-2, 5)$ produces the value-tree $\theta_1 = \texttt{True}\langle <, -2, 5, \texttt{True}\rangle$ introduced in Example 3.3. The operational interpretation $(\!|<|\!)^{\Theta, \sigma}_{\delta, \delta'}$ of $<$ is the following (notice that this interpretation does not depend on $\Theta, \sigma, \delta, \delta'$, since $<$ is a pure mathematical operator):

$$(\!|<|\!)^{\Theta, \sigma}_{\delta, \delta'} = \lambda x . \lambda y . \begin{cases} \texttt{True} & x < y \\ \texttt{False} & \text{otherwise} \end{cases}$$

The value of the whole expression, $\texttt{True}$ (the root of the last subtree of the value-tree), has been computed by using rule [E-B-APP] to evaluate the application $(\!|<|\!)^{\Theta, \sigma}_{\delta, \delta'}(-2, 5)$ of the less-then operator $<$ (the root of the first subtree of the value-tree) to the values -2 (the root of the second subtree of the value-tree) and 5 (the root of the third subtree of the value-tree).

In case $\texttt{e}$ evaluates to a user-defined or anonymous function $\texttt{f}$, rule [E-D-APP] is used: it performs domain restriction $\pi^{\texttt{f}}(\Theta)$ (thus discarding devices that did not apply the same function $\texttt{f}$, for which no consistent information on the application of $\texttt{f}$ is present), then continues the evaluation by substituting the arguments into the body of $\texttt{f}$. We remark that we do not assume that Θ is empty whenever it does not contain δ. In fact, in any round where $\texttt{e}$ evaluates to a function $\texttt{f}$ for the first time on a device, $\texttt{f}(\overline{\texttt{e}})$ will be evaluated with respect to an environment not containing δ but possibly containing other devices (whose $\texttt{e}$ evaluated to $\texttt{f}$ in their previous round of computation).

Example 3.7 (Defined or anonymous function application). Evaluating the expression $\texttt{f}(3)$, where $\texttt{f}$ is the name of the declared function $\texttt{def f(x)\{x + 1\}}$, produces the value-tree $\theta_2 = 4\langle \texttt{f}, 3, 4\langle +, 3, 1, 4\rangle\rangle$ introduced in Example 3.3. The value of the whole expression, 4 (the root of θ_2), which has been computed by using rule [E-D-APP], is the root of the last

subtree of θ_2, which is produced by the evaluation of the expression $3 + 1$ (obtained from the body of f by replacing x with 3). Evaluating the similar expression $f(3)$ where f is the anonymous function $((x) \overset{\tau}{=\!\!>} \{x + 1\})$, produces the same value-tree θ_2 by the same rule [E-D-APP].

Rule [E-REP] implements internal state evolution through computational rounds: on the first firing of a device, $\mathtt{rep}(\mathtt{e_1})\{\mathtt{e_2}\}$ evaluates to $\mathtt{e_2}(\mathtt{e_1})$, then it evaluates to $\mathtt{e_2}(\mathtt{v})$ where $\mathtt{v}$ is the value calculated in the previous round.

Example 3.8 (Time evolution). To illustrate rule [E-REP], as well as computational rounds, we consider the program $\mathtt{rep}(3)\{\mathtt{f}\}$ where $\mathtt{f}$ is the anonymous function $(x) \overset{\tau}{=\!\!>} \{x + 1\}$ introduced in Example 3.7. The first firing of a device δ is performed against the empty tree environment.[6] Therefore, according to rule [E-REP], evaluating $\mathtt{rep}(3)\{\mathtt{f}\}$ produces the value-tree $\theta = 4\langle 3, \theta_2 \rangle$ where $\theta_2 = 4\langle \mathtt{f}, 3, 4\langle +, 3, 1, 4 \rangle \rangle$ is the value-tree (introduced in Example 3.3) produced by evaluating the expression $f(3)$ as described in Example 3.7. The overall result of the firing is the root 4 of θ. Any subsequent firing of the device δ is performed with respect to a value-tree environment Θ that associates to δ the outcome θ of the most recent firing of δ. Therefore, evaluating $\mathtt{rep}(3)\{\mathtt{f}\}$ at the second firing produces the value-tree $\theta' = 5\langle 4, \theta_2' \rangle$ where $\theta_2' = 5\langle \mathtt{f}, 4, 5\langle +, 4, 1, 5 \rangle \rangle$ is the value-tree produced by evaluating the expression $f(4)$, where 4 is the root of θ. Hence, the results of the firings are 4, 5, 6, and so on.

Rules [E-NBR] and [E-NBR-LOC] model device interaction (together with [E-FOLD] which we shall consider later). When an $\mathtt{nbr}$-expression is not evaluated against a neighbour (that is, $\delta' = \delta$), by Rule [E-NBR-LOC] the $\mathtt{nbr}$ operator is discarded and the evaluation continues. Whenever instead an $\mathtt{nbr}$-expression is evaluated against a neighbour (that is, $\delta' \neq \delta$), by Rule [E-NBR] the expression directly evaluates to $\Theta(\delta')$ (which is the value-tree calculated by device δ' in its last computational round for the same expression). Notice that it could be possible that δ' is not in the domain of Θ due to alignment operations performed in subexpressions of the enclosing instance of $\mathtt{foldhood}$. In this case, no rule is applicable and the $\mathtt{nbr}$-expression *fails*, causing δ' to be ignored by the enclosing $\mathtt{foldhood}$ operator (see Rule [E-FOLD]).

Rule [E-FOLD] implements collection and aggregation of results from neighbours, proceeding in the following steps:

- Evaluate the initial value $\mathtt{e_1}$ with respect to the current device obtaining the value-tree θ^1.
- Evaluate the aggregator $\mathtt{e_2}$ with respect to the current device obtaining θ_f with root $\mathtt{f}$.
- Evaluate the body $\mathtt{e_3}$ with respect to the current device $\delta_0 = \delta$, obtaining θ_0 which constitutes the third branch of the overall resulting value-tree, then with respect to every one of the n neighbours, $\delta' \in \mathbf{dom}(\pi_3(\Theta)) \setminus \{\delta\}$, and *consider* only the subset of m neighbours $(0 \leq m \leq n)$ $\delta_1, ..., \delta_m$ *for which the evaluation does not fail* (cf. discussion above and Figure 7), obtaining the value-trees $\theta_1, ..., \theta_m$, respectively.[7] We remark that

---

[6] In the first firing, no messages have been yet received, and therefore the environment is empty. This fact will be later formalised and discussed in the next section on network semantics.

[7] Usually, the aggregator $\mathtt{f}$ is associative and commutative, so that the result of the aggregation does not depend on the order in which the neighbours $\delta' \in \mathbf{dom}(\pi_3(\Theta)) \setminus \{\delta\}$ are considered. To ensure determinism even in the unlikely case of the aggregator $\mathtt{f}$ being not associative and commutative, we assume that the neighbours are considered according to any given total order on device identifiers. Such an order may be, e.g., lexicographical order of the bit representations of the unique device identifiers δ (which may be, e.g., MAC addresses).

the evaluation with respect to neighbours δ' is performed locally, with respect to the locally-available information in Θ.

- Aggregate the values $\rho(\theta_i)$ $(1 \le i \le m)$ computed above together with the initial value $\rho(\theta^1)$ via function $\mathtt{f}$, obtaining the final outcome $\rho(\theta^{m+1})$. Notice that when $m = 0$, the final outcome is the result $\rho(\theta^1)$ of $\mathtt{e}_1$. The aggregation is performed with respect to the current device and the empty environment, since the value-trees of the aggregation process cannot be meaningfully related with one another (and thus are not stored in the final outcome of the computation). In other words, the aggregator $\mathtt{f}$ is forced to be a "pure" function independent of the current device and environment (even though the expression $\mathtt{e}_2$ as a whole might depend on the environment).

The values aggregated by $\mathtt{foldhood}$ exclude the value of $\mathtt{e}_3$ in the current device δ. However, an inclusive folding operation $\mathtt{foldhoodPlusSelf}$ can be encoded as

```
def foldhoodPlusSelf(f, v) { foldhood(v, f, v) }.
```

We also remark that a sequence of nested $\mathtt{foldhood}$-operators (not interleaved by $\mathtt{nbr}$-operators) can lead to an evaluation time which is exponential in the evaluation tree depth.[8]

Failure of evaluation against a neighbour is formalised by means of the auxiliary judgement $\delta, \delta'; \Theta; \sigma \vdash \mathtt{e}$ FAIL defined by the rules in Figure 7. Rules [E-NBR-FAIL] and [E-R-APP-FAIL] model the failure sources, while the other rules model failure propagation.

Example 3.9 (Neighbourhood interaction). To illustrate rules [E-FOLD], [E-NBR] and [E-NBR-LOC], we consider program

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

evaluated in device δ_0 (in which $\mathtt{temperature}() = 10$) with neighbours δ_1 (in which $\mathtt{temperature}() = 15$) and δ_2 (in which $\mathtt{temperature}() = 5$). By Rule [E-FOLD], the three subexpressions of the $\mathtt{foldhood}$-expression are evaluated with respect to δ_0 into the value-trees $\theta^1, \theta_f, \theta_0$ which will constitute the branches of the final tree. The first two of them are $\theta^1 = 2\langle\rangle$ and $\theta_f = +\langle\rangle$, each obtained by Rule [E-VAL]. Then, the third subexpression is evaluated against δ_0, δ_1 and δ_2, obtaining:

$$\theta_0 = 10\langle\mathtt{min}, 10\langle 10\langle\mathtt{temperature}, 10\rangle\rangle, 10\langle\mathtt{temperature}, 10\rangle, 10\rangle,$$
$$\theta_1 = 10\langle\mathtt{min}, 15\langle 15\langle\mathtt{temperature}, 15\rangle\rangle, 10\langle\mathtt{temperature}, 10\rangle, 10\rangle,$$
$$\theta_2 = 5\langle\mathtt{min}, 5\langle 5\langle\mathtt{temperature}, 5\rangle\rangle, 10\langle\mathtt{temperature}, 10\rangle, 5\rangle$$

the first one (θ_0) obtained through three applications of Rule [E-B-APP] and one of Rule [E-NBR-LOC], and the other two $(\theta_1$ and $\theta_2)$ obtained through three applications of Rule [E-B-APP] and one of Rule [E-NBR]. The roots of value-trees θ_1 and θ_2 are then combined through operator $+$, together with the initial value 2, for a total result of $2 + 10 + 5 = 17$ which is the root of the final value-tree

---

[8]In actual implementations, the outcome of $\mathtt{foldhood}$ and $\mathtt{rep}$ subexpressions can be "memoised" in order to prevent subsequent re-evaluation (since such expressions are independent of the neighbour against which are evaluated). This addresses the performance issues of nested $\mathtt{foldhood}$-operators.

$17\langle 2, +, \theta_0 \rangle$.   Consider the following graphical representation of the value-trees involved.
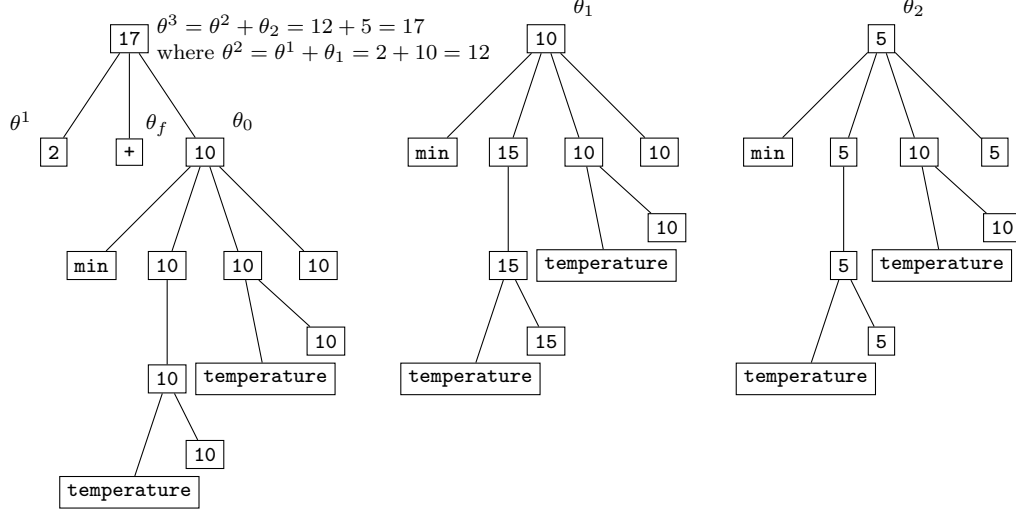
Rules [E-VAL], [E-REP], [E-FOLD] are independent of the neighbour δ' against which the expression is computed (since δ' does not occur in the premises of those rules). Rules [E-B-APP] and [E-D-APP] simply pass δ' through, allowing subexpressions to make use of it (including evaluation of built-in relational sensors $\mathbf{r}$). The neighbour device δ' is then non-trivially exploited only in rules [E-NBR], [E-NBR-LOC], [E-NBR-FAIL] and [E-R-APP-FAIL].

We say that a neighbour is *considered* by the evaluation of a `foldhood`-expression to mean that it contributes to the result of the expression. Because of the interplay between neighbourhood interaction and branching (i.e., function call) only a subset of the neighbourhood of a device might be considered by a `foldhood`-expression.

Example 3.10 (Combining time evolution with neighbourhood interaction: the gradient)**.** Consider the *gradient* function from Example 3.1.

```
def gradient(source, metric) { // : (bool, ()->num) -> num
  rep(PositiveInfinity){ (distance) => {
    mux(source, 0.0,
      foldhood(PositiveInfinity, min, nbr{distance} + metric())
    )
  }}
}
```

The `gradient` function computes the field of minimum distances (according to `metric`) from devices where `source` is `True`. It uses `rep` to keep track of the local gradient value, which is computed by looking at the corresponding value in the neighbourhood. If `source` is locally `True`, then the value is merely `0.0`, since it means that the device is a source; otherwise, the gradient value is obtained by folding over neighbours (via `foldhood`) to collect the minimum value of `nbr{distance}+metric()`. The repeated application of such a function, together with actual communication (formally covered in Section 3.4) consisting of the `nbr` evaluations of neighbours, makes the output field eventually converge to the correct value (minimum distances from sources). The gradient is a fundamental building block for collective adaptive behaviour. It is amenable to various implementations [ACDV17] and plays a crucial role in higher-level patterns [CPVN19], as also shown in the case study of Section 6.

Example 3.11 (Neighbourhood interaction and branching). In order to illustrate the alignment process, guiding neighbour interaction through branching statements, consider the expression for a *gradient avoiding obstacles*, introduced in Example 3.1.

```
if (isObstacle) { PositiveInfinity } { gradient(isSource) }  // : num
```

Expanding the syntactic sugar, the `if` statement corresponds to the execution of a different anonymous function depending on the value of `isObstacle`:

```
mux( isObstacle, () => {PositiveInfinity}, () => {gradient(isSource)} )()
```

Assume that device δ_0 evaluates this program with respect to $\Theta = \{\delta_0 \mapsto \theta_0, \delta_1 \mapsto \theta_1, \delta_2 \mapsto \theta_2\}$, where `isObstacle` is true in δ_2 and false on the other devices. Thus, the execution of the `mux` statement produces $f_\perp = () \Rightarrow \{\texttt{gradient(isSource)}\}$ on δ_0 and δ_1, while it produces $f_\top = () \Rightarrow \{\texttt{PositiveInfinity}\}$ on δ_2.

The evaluation of the main expression is performed through rule [E-D-APP]. First, the function to be applied is computed as the result of the `mux` expression. Then, the body `gradient(isSource)` is computed with respect to the environment $\pi^{f_\perp}(\Theta) = \{\delta_0 \mapsto \pi_2(\theta_0), \delta_1 \mapsto \pi_2(\theta_1)\}$: the value-tree of device δ_2 is removed since it corresponded to the evaluation of $f_\top$. The evaluation of `gradient(isSource)` will then require the evaluation of the `foldhood` expression, in which only devices δ_0 and δ_1 will be considered (since δ_2 has already been discarded).

3.4. **Operational Semantics: Network Semantics.** We now provide an operational semantics for the evolution of whole networks, namely, for modelling the distributed evolution of computational fields over time. The semantics is given as a nondeterministic, small-step transition system on network configurations N. This semantics has already been given for HFC in [ABD+20], with the only difference of referring to the HFC device semantics instead of the NC device semantics. Figure 8 (top) defines key syntactic elements to this end:

- Ψ is a computational field (called *value-tree field*) that models the overall state of the computation as a map from device identifiers to the value-tree environments that are locally stored in the corresponding devices.
- α is an *activation predicate* specifying whether each device is currently activated (i.e., is performing a computation round).
- *Stat* (a pair of value-tree field and activation predicate) models the overall computation *status*.
- $\rightarrowtail$ models network *topology* as a directed neighbouring graph, i.e. a reflexive neighbouring relation $\rightarrowtail \subseteq \mathbf{D} \times \mathbf{D}$ so that $\delta \rightarrowtail \delta$ for each $\delta \in \mathbf{D}$.
- Σ models (distributed) *sensor state*, as a map from device identifiers to (local) sensors representations (i.e., sensor name/value maps denoted as σ).
- *Env* (a pair of topology and sensor state) models the network *environment*.
- N (a pair of status and environment) models a whole *network configuration*.

We use the following notation for maps. Let $\bar{x} \mapsto y$ denote a map sending each element in the sequence $\bar{x}$ to the same element y. Let $m_0[m_1]$ denote the map with domain $\mathbf{dom}(m_0) \cup \mathbf{dom}(m_1)$ coinciding with m_1 in the domain of m_1 and with m_0 otherwise. Let $m_0[\![m_1]\!]$ (where m_i are maps to maps) denote the map with the *same domain* as m_0 made of $x \mapsto m_0(x)[m_1(x)]$ for all x in the domain of m_1, $x \mapsto m_0(x)$ otherwise. The notation $F_\delta(\cdot)$ used in rule [N-COMP], Figure 8 (bottom), models a filtering operation that clears out

---

System configurations and action labels:

Ψ	$::=$	$\overline{\delta} \mapsto \overline{\Theta}$	value-tree field
α	$::=$	$\overline{\delta} \mapsto \overline{a}$ with $a \in \{\mathtt{false}, \mathtt{true}\}$	activation predicate
$Stat$	$::=$	$\langle \Psi, \alpha \rangle$	status
$\rightarrowtail$	$::=$	$\langle \overline{\delta}, \overline{\delta}' \rangle$	topology
Σ	$::=$	$\overline{\delta} \mapsto \overline{\sigma}$	sensor state
Env	$::=$	$\langle \rightarrowtail, \Sigma \rangle$	environment
N	$::=$	$\langle Env; Stat \rangle$	network configuration
act	$::=$	$\delta+ \mid \delta- \mid env$	action label

Environment well-formedness:

$WFE(\langle \rightarrowtail, \Sigma \rangle)$ holds iff $\{\langle \delta, \delta \rangle \mid \delta \in \mathbf{D}\} \subseteq \rightarrowtail \subseteq \mathbf{D} \times \mathbf{D}$ where $\mathbf{D} = \mathbf{dom}(\Sigma)$

Transition rules for network evolution: $\boxed{N \xrightarrow{act} N}$

[N-COMP] $\dfrac{\alpha(\delta) = \mathtt{false} \quad \Theta' = F_\delta(\Psi(\delta)) \quad \delta, \delta; \Theta'; \Sigma(\delta) \vdash \mathtt{e_{main}} \Downarrow \theta \quad \Theta = \Theta'[\delta \mapsto \theta]}{\langle \langle \rightarrowtail, \Sigma \rangle; \langle \Psi, \alpha \rangle \rangle \xrightarrow{\delta+} \langle \langle \rightarrowtail, \Sigma \rangle; \langle \Psi[\delta \mapsto \Theta], \alpha[\delta \mapsto \mathtt{true}] \rangle \rangle}$

[N-SEND] $\dfrac{\alpha(\delta) = \mathtt{true} \quad \overline{\delta} = \{\delta' \mid \delta \rightarrowtail \delta'\} \quad \theta = \Psi(\delta)(\delta) \quad \Theta = \delta \mapsto \theta}{\langle \langle \rightarrowtail, \Sigma \rangle; \langle \Psi, \alpha \rangle \rangle \xrightarrow{\delta-} \langle \langle \rightarrowtail, \Sigma \rangle; \langle \Psi[\![\overline{\delta} \mapsto \Theta]\!], \alpha[\delta \mapsto \mathtt{false}] \rangle \rangle}$

[N-ENV] $\dfrac{WFE(Env') \quad Env' = \langle \rightarrowtail, \overline{\delta} \mapsto \overline{\sigma} \rangle \quad \Psi_0 = \overline{\delta} \mapsto \emptyset \quad \alpha_0 = \overline{\delta} \mapsto \mathtt{false}}{\langle Env; \Psi, \alpha \rangle \xrightarrow{env} \langle Env'; \Psi_0[\Psi], \alpha_0[\alpha] \rangle}$

FIGURE 8. Small-step operational semantics for network evolution.

old stored value-trees from $\Psi(\delta)$, implicitly based on space/time tags.[9] Notice that this mechanism allows messages to persist across rounds.

We define network operational semantics in terms of small-steps transitions $N \xrightarrow{act} N'$ of three kinds: firing starts on a given device (for which act is $\delta+$ where δ is the corresponding device identifier), firing ends and messages are sent on a given device (for which act is $\delta-$), and environment changes, where act is the special label env. This is formalised in Figure 8 (bottom).

Rule [N-COMP] (available for sleeping devices, i.e., with $\alpha(\delta) = \mathtt{false}$, and setting them to executing, i.e., $\alpha(\delta) = \mathtt{true}$) models a computation round at device δ: it takes the local value-tree environment filtered out of old values $\Theta' = F_\delta(\Psi(\delta))$; then by the single device semantics it obtains the device's value-tree θ, which is used to update the system configuration of δ to $\Theta = \Theta'[\delta \mapsto \theta]$. Notice that expression $\mathtt{e_{main}}$ is always evaluated against the device δ itself (that is, against no neighbour), and that local sensors $\Sigma(\delta)$ are used by the auxiliary function $(\![\mathtt{b}]\!)_{\delta, \delta'}^{\Theta, \Sigma(\delta)}$ that gives the semantics to the built-in functions. Furthermore, although this rule updates a device's system configuration instantaneously, it models computations taking an arbitrarily long time, since the update is not visible until the following rule [N-SEND]. Notice also that all values used to compute θ are locally available

---

[9]For example, the filter may remove value-trees that were stored before $t - \Delta t$, where t is the time of the current firing and Δt is a decay parameter of the filter.

(at the beginning of the computation), thus allowing for a fully-distributed implementation without global knowledge.

Rule [N-SEND] (available for running devices with $\alpha(\delta) = \texttt{true}$, and setting them to non-running) models the message sending happening at the end of a computation round at a device δ. It takes the local value-tree $\theta = \Psi(\delta)(\delta)$ computed by last rule [N-COMP], and uses it to update neighbours' $\bar{\delta}$ values of $\Psi(\bar{\delta})$. Notice that the usage of α ensures that occurrences of rules [N-COMP] and [N-SEND] for a device are alternated.

Rule [N-ENV] takes into account the change of the environment to a new *well-formed* environment Env'—environment well-formedness is specified by the predicate $WFE(Env)$ in Figure 8 (middle)—thus modelling node mobility as well as changes in environmental parameters. Let $\bar{\delta}$ be the domain of Env'. We first construct a value-tree field Ψ_0 and an activation predicate α_0 associating to all the devices of Env' the empty context $\emptyset$ and the $\texttt{false}$ activation. Then, we adapt the existing value-tree field Ψ and activation predicate α to the new set of devices: $\Psi_0[\Psi]$, $\alpha_0[\alpha]$ automatically handles removal of devices, mapping of new devices to the empty context and $\texttt{false}$ activation, and retention of existing contexts and activation in the other devices. We remark that this rule is also used to model communication failure as topology changes.

Remark 3.12 (Locality of interactions). We remark that although the network semantics is given from a *global* perspective, it only allows for interactions that are local in nature. Rule [N-COMP] can be understood as happening on device δ, with respect to locally-available knowledge and without interactions with other devices. Rule [N-SEND] represents the broadcast of a value locally available on δ, which is then received by neighbour devices and stored in their local storage. Rule [N-ENV] does not represent an interaction at all, but only a change in the set of possible interactions.

Example 3.13 (Network evolution). Consider the program in Example 3.9:

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

and let $\theta^n = n\langle\texttt{min}, n\langle n\langle\texttt{temperature}, n\rangle\rangle, n\langle\texttt{temperature}, n\rangle, n\rangle$ be the result of evaluation of $\texttt{min}(\texttt{nbr}\{\texttt{temperature()}\}, \texttt{temperature()})$ in a device where $\texttt{temperature()} = n$ (with respect to the device itself as neighbour).

We start from a configuration $N_0 = \langle\langle\rightarrowtail, \Sigma\rangle; \langle\Psi_0, \alpha_0\rangle\rangle$ with three devices $\bar{\delta}$, so that $\rightarrowtail = \{(\delta_i, \delta_j) \mid i, j \leq 3\}$ (all devices are connected), $\Psi_0 = \bar{\delta} \mapsto \emptyset$ (devices do not hold any information), $\alpha_0 = \bar{\delta} \mapsto \texttt{false}$ (devices are not computing) and

$$\Sigma = \delta_1 \mapsto \{t = 10\}, \delta_2 \mapsto \{t = 15\}, \delta_3 \mapsto \{t = 5\}$$

(temperatures are as in Example 3.9).

After transitions $N_0 \xrightarrow{\delta_2^+} N \xrightarrow{\delta_2^-} N_1$, the computational field Ψ_0 is updated by sending the result $\theta_0 = 2\langle 2, \texttt{+}, \theta^{15}\rangle$ of the computation of δ_2 (with respect to its empty environment) to every device, obtaining $\Psi_1 = \bar{\delta} \mapsto \{\delta_2 \mapsto \theta_0\}$. Then, other transitions take place: $N_1 \xrightarrow{\delta_3^+} N \xrightarrow{\delta_3^-} N_2$, where Ψ_1 is further updated with the result $\theta_1 = 7\langle 2, \texttt{+}, \theta^5\rangle$ of the computation of δ_3 (with respect to the information received from δ_2), obtaining $\Psi_2 = \bar{\delta} \mapsto \{\delta_2 \mapsto \theta_0, \delta_3 \mapsto \theta_1\}$. Finally, transitions $N_2 \xrightarrow{\delta_1^+} N \xrightarrow{\delta_1^-} N_3$ happen as described in Example 3.9, producing $\Psi_3 = \bar{\delta} \mapsto \{\delta_1 \mapsto \theta_2, \delta_2 \mapsto \theta_0, \delta_3 \mapsto \theta_1\}$ where $\theta_2 = 17\langle 2, \texttt{+}, \theta^{10}\rangle$.

Lastly, a transition $N_3 \xrightarrow{env} N_4$ may happen, lowering temperatures, deleting device δ_2, inserting device δ_4, and disconnecting device δ_1 from device δ_3. The result is configuration

$N_4 = \langle \langle \rightarrowtail', \Sigma' \rangle; \langle \Psi_4, \alpha_4 \rangle \rangle$ where:

$\quad \rightarrowtail' = \langle \delta_1, \delta_4 \rangle, \langle \delta_3, \delta_4 \rangle, \langle \delta_4, \delta_1 \rangle, \langle \delta_4, \delta_3 \rangle$

$\quad \Sigma' = \delta_1 \mapsto \{t = 9\}, \delta_1 \mapsto \{t = 4\}, \delta_4 \mapsto \{t = 1\}$

$\quad \Psi_4 = \delta_1 \mapsto \{\delta_1 \mapsto \theta_2, \delta_2 \mapsto \theta_0, \delta_3 \mapsto \theta_1\}, \delta_3 \mapsto \{\delta_1 \mapsto \theta_2, \delta_2 \mapsto \theta_0, \delta_3 \mapsto \theta_1\}, \delta_4 \mapsto \emptyset,$

$\quad \alpha_4 = \delta_1 \mapsto \texttt{false}, \delta_3 \mapsto \texttt{false}, \delta_4 \mapsto \texttt{false}.$

Notice that devices δ_1, δ_3 are not aware yet of the disappearance of δ_2, nor of their disconnection. When one of them will fire, the filter $F(\cdot)$ may be able to remove the obsolete values from the corresponding value-tree environments.

3.5. **Type Preservation in NC.** In this section we show that the evaluation rules for NC are deterministic and preserve types, provided that the value-tree environment used for the evaluation is coherent with the expression being evaluated according to the following definition.

Definition 3.14 (Well Formed Value Tree). Given a closed expression $\texttt{e}$, a local-type-scheme environment $\mathcal{D}$, a type environment $\mathcal{A} = \overline{\texttt{x}} : \overline{T}$, and a type T such that $\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T$ holds, the set $WFVT(\mathcal{D}; \mathcal{A}; \texttt{e})$ of the *well-formed value-trees* for $\texttt{e}$ is inductively defined as follows. $\theta \in WFVT(\mathcal{D}; \mathcal{A}; \texttt{e})$ if and only if $\texttt{v} = \rho(\theta)$ has type T (i.e. $\mathcal{D}; \emptyset \vdash \texttt{v} : T$) and

- if $\texttt{e}$ is a value, θ is of the form $\texttt{v}\langle\rangle$;
- if $\texttt{e} = \texttt{nbr}\{\texttt{e}_1\}$, θ is of the form $\texttt{v}\langle\theta_1\rangle$ where $\theta_1 \in WFVT(\mathcal{D}; \mathcal{A}; \texttt{e}_1)$;
- if $\texttt{e} = \texttt{rep}(\texttt{e}_1)\{\texttt{e}_2\}$, θ is of the form $\texttt{v}\langle\theta_1, \theta_2\rangle$ where $\theta_1 \in WFVT(\mathcal{D}; \mathcal{A}; \texttt{e}_1)$ and $\theta_2 \in WFVT(\mathcal{D}; \mathcal{A}, \texttt{x} : T; \texttt{e}_2(\texttt{x}))$;
- If $\texttt{e} = \texttt{foldhood}(\texttt{e}_1, \texttt{e}_2, \texttt{e}_3)$, θ is of the form $\texttt{v}\langle\theta_1, \theta_2, \theta_3\rangle$ where $\overline{\theta} \in WFVT(\mathcal{D}; \mathcal{A}; \overline{\texttt{e}})$;
- if $\texttt{e} = \texttt{e}'(\overline{\texttt{e}})$ and $\mathcal{D}; \mathcal{A} \vdash \texttt{e}' : T'$, $\mathcal{D}; \mathcal{A} \vdash \overline{\texttt{e}} : \overline{T}$, then θ is of the form $\texttt{v}\langle\theta', \overline{\theta}, \theta''\rangle$ where $\theta' \in WFVT(\mathcal{D}; \mathcal{A}; \texttt{e}')$, $\overline{\theta} \in WFVT(\mathcal{D}; \mathcal{A}; \overline{\texttt{e}})$, and either:
  - $\texttt{f} = \rho(\theta')$ is a built-in function and $\theta'' = \texttt{v}\langle\rangle$,
  - $\texttt{f}$ is not a built-in function and $\theta'' \in WFVT(\mathcal{D}; \mathcal{A}, args(\texttt{f}) : \overline{T}; body(\texttt{f}))$.

Similarly, the set of *well-formed value-tree environments* $WFVTE(\mathcal{D}; \mathcal{A}; \texttt{e})$ is the set of $\Theta = \overline{\delta} \mapsto \overline{\theta}$ such that $\overline{\theta} \in WFVT(\mathcal{D}; \mathcal{A}; \texttt{e})$.

In other words, the above definition demands value-trees to be plausible outcomes of the evaluation of $\texttt{e}$.

Lemma 3.15 (Computation Determinism). *Let $\texttt{e}$ be a well-typed closed expression and $\Theta \in WFVTE(\mathcal{D}; \mathcal{A}; \texttt{e})$. Then for all device identifiers δ, δ' and sensor state σ:*

(1) *$\delta, \delta; \Theta; \sigma \vdash \texttt{e}$ FAIL cannot hold.*
(2) *There is at most one derivation of the kind $\delta, \delta'; \Theta; \sigma \vdash \texttt{e} \Downarrow \theta$ or $\delta, \delta'; \Theta; \sigma \vdash \texttt{e}$ FAIL.*

Proof. The proof follows from standard induction on rules (see Appendix B.1). $\qquad\square$

By this lemma, evaluation does not result on FAIL when it is performed relative to the current device (as it is the case for main expressions), and rules are deterministic. Furthermore, the evaluation rules respect the types given in Figure 4, provided that the *built-in interpretations respect the given types*. Formally, given $\texttt{b}$ such that $\mathcal{B}; \emptyset \vdash \texttt{b} : \overline{T} \rightarrow T$ and any $\mathcal{B}; \emptyset \vdash \overline{\texttt{v}} : \overline{T}$, $\Theta = \overline{\delta} \mapsto \overline{\texttt{v}}'\langle\rangle$ with $\mathcal{B}; \emptyset \vdash \overline{\texttt{v}}' : T$, $\delta' \in \{\delta, \overline{\delta}\}$, then we require $(\!|\texttt{b}|\!)_{\Theta,\sigma}^{\delta,\delta'}$ to be a value of type T.

Theorem 3.16 (Type Preservation). *Assume that the interpretation of built-in operators respects the given types. Let $\mathcal{A} = \overline{\mathrm{x}} : \overline{T}$ and $\mathcal{D}; \emptyset \vdash \overline{\mathrm{v}} : \overline{T}$, so that $length(\overline{\mathrm{v}}) = length(\overline{\mathrm{x}})$. If $\mathcal{D}; \mathcal{A} \vdash \mathrm{e} : T$, $\Theta \in WFVTE(\mathcal{D}; \mathcal{A}; \mathrm{e})$ and $\delta, \delta'; \Theta; \sigma \vdash \mathrm{e}[\overline{\mathrm{x}} := \overline{\mathrm{v}}] \Downarrow \theta$, then $\theta \in WFVT(\mathcal{D}; \mathcal{A}; \mathrm{e})$.*

Notice that, since the evaluation of e produces a value-tree which is coherent with e, the value-tree environment Θ can be proved to be coherent with the main expression by induction on the network evolution. Furthermore, observe that the typing rules (in Figure 4) and the evaluation rules (in Figure 6 and 7) are syntax directed. Then the proof can be carried out by induction on the derivation length for $\delta, \delta'; \Theta; \sigma \vdash \mathrm{e}[\overline{\mathrm{x}} := \overline{\mathrm{v}}] \Downarrow \theta$, while using the following standard lemmas.

Lemma 3.17 (Substitution). *Let $\mathcal{A} = \overline{\mathrm{x}} : \overline{T}$, $\mathcal{B}; \emptyset \vdash \overline{\mathrm{v}} : \overline{T}$. If $\mathcal{D}; \mathcal{A} \vdash \mathrm{e} : T$, then $\mathcal{D}; \emptyset \vdash \mathrm{e}[\overline{\mathrm{x}} := \overline{\mathrm{v}}] : T$.*

Proof. Straightforward by induction on application of the typing rules for expressions in Figure 4. □

Lemma 3.18 (Weakening). *Let $\mathcal{D}' \supseteq \mathcal{D}$, $\mathcal{A}' \supseteq \mathcal{A}$ be such that $\boldsymbol{dom}(\mathcal{D}') \cap \boldsymbol{dom}(\mathcal{A}') = \emptyset$. If $\mathcal{D}; \mathcal{A} \vdash \mathrm{e} : T$, then $\mathcal{D}'; \mathcal{A}' \vdash \mathrm{e} : T$.*

Proof. Straightforward by induction on application of the typing rules for expressions in Figure 4. □

Proof of Theorem 3.16. We proceed by induction on the derivation length.

The fact that $\rho(\theta)$ has type T can be verified by matching step-by-step every rule in Figure 6 with the corresponding rule in Figure 4, while using the inductive hypothesis and two further assumptions: for rule [E-B-APP], that built-in functions b respect the given types; for rules [E-NBR] and [E-REP], that Θ is coherent with e.

Finally, the fact that θ has the required sub-trees follows by inductive hypothesis since every rule in Figure 6 respects the corresponding row of Definition 3.14, together with the fact that Θ is coherent with e (for rules [E-NBR] and [E-REP] only). □

4. NC vs HFC

In this section, we provide a formal account of the relationship between NC and the HFC minimal core calculus for Aggregate Computing [AVD+19].

In Section 4.1, we give an informal overview of the relationship between the two calculi. In Section 4.2 we recollect HFC. In Section 4.3, we define a fragment of HFC, which we call HFC′, aimed at ensuring that each HFC′ program is an NC program that behaves in the same way. In Section 4.4, we define the fragment of NC which corresponds to HFC′, which we call call NC′. Then, in Section 4.5 we prove the equivalence between NC′ and HFC′. Finally, in Section 4.6, we point out that NC provides a different "flavour" of field computation with respect to HFC, though without losing practical expressiveness.
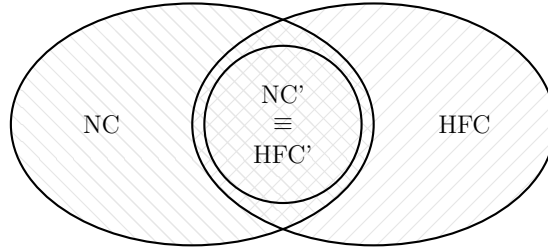
Figure 9. Pictorial representation of relationship between NC, HFC and their fragments NC′ and HFC′.

4.1. **Overview.** NC and HFC are languages with a similar purpose and syntax, but differently structured semantics. Both languages are Turing-complete (in fact, *space-time universal* [ABDV18]), thus they can both express all distributed computations, although possibly with structurally different programs.

As the syntax of NC can be mapped to/from that of HFC, a question naturally arises: what are the distributed computations that are expressed in both languages *by the same program*? The answer to this question, detailed in the remainder of this section, is summarised in Figure 9. Not all programs are interpreted to the same behaviour in NC and HFC, however, the (subset of) programs with identical behaviour in NC and HFC can be identified. Through Lemma 4.3 in Section 4.3, we define a restricted language HFC′ and show that it is indeed a subset of HFC. Similarly, in Theorem 4.5 we define NC′ while ensuring that it is a subset of NC. Finally, Theorem 4.7 shows that HFC′ coincides with NC′, concluding the characterisation of this common fragment of programs with identical behaviour.

We remark that HFC′/NC′ does *not* coincide with the "intersection" between HFC and NC: there are programs outside HFC′/NC′ which nonetheless result in the same behaviour in both HFC and NC. Still, we will show in Section 4.6 that most interesting programs do fit inside HFC′/NC′, or can be refactored to fit into it through a small set of simple *rewriting strategies*. In fact, we are not aware of any practically relevant program that cannot be refactored into HFC′/NC′ through the proposed strategies. We also remark that as HFC′/NC′ is Turing-complete, some equivalent form in HFC′/NC′ must exist for any given distributed program. Overall, these results show that NC provides a different "flavour" of field computation with respect to HFC, without losses in expressiveness.

4.2. **A Quick Recollection of HFC.** The syntax of HFC programs (according to its original presentation [AVD+19]) is the same of NC programs (in Figure 3), with a richer syntax of values and two other minor differences: the update function e_2 in a `rep` construct `rep(e_1){e_2}` is required to be an anonymous function $e_2 = (x) \Rightarrow \{e'\}$,[10] and the language construct for `foldhood` is replaced by a built-in with the same meaning, so it does not appear in the syntax. In the richer syntax of HFC values (given in Figure 10), values are divided into *local values*, which are the values of NC, and *neighbouring (field) values*, which not allowed to appear in source code and arise at runtime. Neighbouring values are maps $\overline{\delta} \mapsto \overline{\ell}$ from device identifiers to local values. In HFC, they are produced by evaluating the `nbr` construct and returned by some built-in functions.

_____

[10]We remark that this difference is historical in nature: HFC could be straightforwardly extended to allow for general update function expressions.

$$
\begin{array}{llll}
\texttt{v} & ::= & \ell \;\big|\; \boxed{\phi} & \text{value} \\[4pt]
\phi & ::= & \boxed{\overline{\delta \mapsto \overline{\ell}}} & \text{neighbouring (field) value} \\[4pt]
\ell & ::= & \texttt{c}(\overline{\ell}) \;\big|\; \texttt{f} & \text{local value} \\[4pt]
\texttt{f} & ::= & \texttt{b} \;\big|\; \texttt{d} \;\big|\; (\overline{\texttt{x}}) \texttt{=>} \{\texttt{e}\} & \text{function value}
\end{array}
$$

FIGURE 10. Syntax of HFC values – extensions with respect to NC values are highlighted in grey.

The Hindley-Milner type system for HFC [AVD$^+$19] is given in Figure 11 (excluding the grey rule [T-FOLD]), and distinguishes between types for *local* values from those that are not (namely, neighbouring types F for neighbouring values), as well as between types that are allowed to be *returned* by functions from those that are not. This induces four different type categories: types T, local types L, return types R, and local return types S—they are illustrated at the bottom of Figure 10. The main restrictions enforced by this type system in order to ensure the *domain alignment* property[11] are:

- anonymous functions cannot capture variables of neighbouring type;
- `rep` statements are demanded to have a local return type;
- neighbouring types can only be built from local return types S (i.e., $F = \texttt{field}(S)$), since neighbouring values need to be aggregated and this is possible only for return types, and avoiding "neighbouring values of neighbouring values" which may lead to unintentionally heavy computations;
- types of the form $(\overline{T}) \to F$ (functions returning neighbouring values) are not return types. Thus, functions of type $(\overline{T}) \to F$ are used almost as in a first-order language. In particular, there is no way to write a non-constant expression `e` evaluating to such a function.

The HFC operational semantics [AVD$^+$19] is given as a transition system analogous to that in Section 3.4, but based on a different judgement for the device operational semantics $\delta; \Theta; \sigma \vdash \texttt{e}_{\texttt{main}} \Downarrow \theta$. For sake of completeness, we report the details of the HFC operational device semantics in Appendix A. In the remainder of this paper, we assume that the built-in operators of HFC always include:

- `consthood(v)`, which returns a neighbouring field value constantly equal to its (local) input `v`;
- `map(f, `$\overline{\phi}$`)`, which applies a function `f` with local inputs and outputs (of any ariety) pointwise to neighbouring field values $\overline{\phi}$;
- `foldhood(`ℓ`, f, `ϕ`)`, which collapses a field ϕ and starting value ℓ via an aggregator `f` (exactly as in NC).

Furthermore, we use `if(e`$_1$`){e`$_2$`}{e`$_3$`}` as short for `mux(e`$_1$`, () =>e`$_2$`, () =>e`$_3$`)()`, as in NC.

4.3. **The HFC$'$ Fragment of HFC.** HFC$'$ is obtained by adding the following two custom restrictions, on how neighbouring field values can be processed, to the Hindley-Milner type system for HFC [AVD$^+$19]:

R1 : Built-in functions need to have local arguments, except for the built-ins `map` and `foldhood`.

---

[11]Domain alignment holds iff the domain of neighbouring values ϕ obtained from expressions `e` is equal to the set of all neighbours which computed the same `e` in their previous evaluation round.

Types:

$$T \quad ::= \quad t \mid R \mid L \qquad\qquad\qquad \text{type}$$
$$L \quad ::= \quad l \mid S \mid (\overline{T}) \to R \qquad\qquad \text{local type}$$
$$R \quad ::= \quad r \mid S \mid F \qquad\qquad\qquad \text{return type}$$
$$S \quad ::= \quad s \mid B \mid (\overline{T}) \to S \qquad \text{local return type}$$
$$F \quad ::= \quad \texttt{field}(S) \qquad\qquad\qquad \text{neighbouring type}$$

Local type schemes:

$$LS \quad ::= \quad \forall \overline{t}\overline{l}\,\overline{r}\overline{s}.L \qquad\qquad \text{local type scheme}$$

Expression typing: $\boxed{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : T}$

$$[\text{T'-VAR}] \quad \frac{}{\mathcal{D}; \mathcal{A}, \texttt{x} : T \vdash \texttt{x} : T} \qquad\qquad [\text{T'-DAT}] \quad \frac{S'[\overline{s} := \overline{S}''] = (\overline{S}) \to S \quad \mathcal{D}; \mathcal{A} \vdash \overline{\ell} : \overline{S}}{\mathcal{D}, \texttt{c} : \forall \overline{s}.S'; \mathcal{A} \vdash \texttt{c}(\overline{\ell}) : S}$$

$$[\text{T'-A-FUN}] \quad \frac{\overline{\texttt{y}} = \mathbf{FV}((\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \{\texttt{e}\}) \quad \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{y}} : \overline{L} \quad \mathcal{D}; \mathcal{A}, \overline{\texttt{x}} : \overline{T} \vdash \texttt{e} : R}{\mathcal{D}; \mathcal{A} \vdash (\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}} \{\texttt{e}\} : (\overline{T}) \to R}$$

$$[\text{T'-N-FUN}] \quad \frac{\texttt{f} \text{ is a (built-in or declared) function}}{\mathcal{D}, \texttt{f} : \forall \overline{t}\overline{l}\,\overline{r}\overline{s}.L; \mathcal{A} \vdash \texttt{f} : L[\overline{t} := \overline{T}, \ \overline{l} := \overline{L}, \ \overline{r} := \overline{R}, \ \overline{s} := \overline{S}]}$$

$$[\text{T'-APP}] \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : (\overline{T}) \to R \quad \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{e}} : \overline{T}}{\mathcal{D}; \mathcal{A} \vdash \texttt{e}(\overline{\texttt{e}}) : R}$$

$$[\text{T'-REP}] \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : S \quad \mathcal{D}; \mathcal{A} \vdash (\texttt{x}) \texttt{=>} \{\texttt{e}_2\} : S \to S}{\mathcal{D}; \mathcal{A} \vdash \texttt{rep}(\texttt{e}_1)\{(\texttt{x}) \texttt{=>} \{\texttt{e}_2\}\} : S} \qquad [\text{T'-NBR}] \quad \frac{\mathcal{D}; \mathcal{A} \vdash \texttt{e} : S}{\mathcal{D}; \mathcal{A} \vdash \texttt{nbr}\{\texttt{e}\} : \texttt{field}(S)}$$

$$[\text{T'-FOLD}] \quad \frac{\begin{array}{ll} \mathcal{D}; \mathcal{A} \vdash \texttt{e}_1 : S & \mathcal{D}; \mathcal{A} \vdash \texttt{e}_3 : \texttt{field}(S) \text{ or } S \\ \mathcal{D}; \mathcal{A} \vdash \texttt{e}_2 : (S, S) \to S & \mathcal{D}; \mathcal{A} \vdash \overline{\texttt{x}} : \overline{L} \quad \text{where } \overline{\texttt{x}} = \mathbf{FV}(\texttt{e}_3) \end{array}}{\mathcal{D}; \mathcal{A} \vdash \texttt{foldhood}(\texttt{e}_1, \texttt{e}_2, \texttt{e}_3) : S}$$

Function typing: $\boxed{\mathcal{D} \vdash \texttt{F} : LS}$

$$[\text{T'-FUNCTION}] \quad \frac{\mathcal{D}, \texttt{d} : (\overline{T}) \to R; \overline{\texttt{x}} : \overline{T} \vdash \texttt{e} : R \quad \overline{t}\overline{l}\,\overline{r}\overline{s} = \mathbf{FTV}((\overline{T}) \to R)}{\mathcal{D} \vdash \texttt{def d}(\overline{\texttt{x}})\{\texttt{e}\} : \forall \overline{t}\overline{l}\,\overline{r}\overline{s}.(\overline{T}) \to R}$$

Program typing: $\boxed{\mathcal{D}_0 \vdash \texttt{P} : T}$

$$[\text{T'-PROGRAM}]$$
$$\frac{\begin{array}{lll} \texttt{F}_i = (\texttt{def d}_i(\_) \ \_) & \mathcal{D}_{i-1} \vdash \texttt{F}_i : LS_i & \mathcal{D}_i = \mathcal{D}_{i-1}, \texttt{d}_i : LS_i \quad (i \in 1..n) \\ \mathcal{D}_n; \emptyset \vdash \texttt{e} : T & & \end{array}}{\mathcal{D}_0 \vdash \texttt{F}_1 \cdots \texttt{F}_n \ \texttt{e} : T}$$
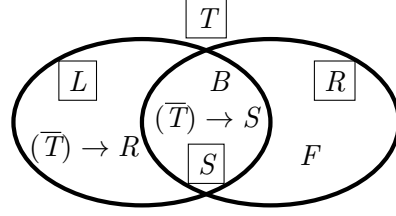
Figure 11. Hindley-Milner typing for HFC$'$ and NC$'$ expressions, function declarations, and programs – differences with HFC typing are highlighted in grey.

R2 : Expressions of neighbouring type can only be aggregated to local values with a `foldhood` operator if they do not capture variables of neighbouring types; so that, e.g., aggregating arguments of neighbouring type is never allowed.

Example 4.1 (About restriction R1). In order to show the rationale behind Restriction R1, consider a built-in function `sorthood` rearranging values $\phi(\delta)$ relative to neighbours in increasing order of neighbour identifier δ, thus effectively mixing up values relative to different neighbours. Formally, applying this function to a neighbouring value $\phi = \overline{\delta} \mapsto \overline{\ell}$ (assuming $\delta_1 \leq \ldots \leq \delta_n$), we obtain the neighbouring value $\phi' = \delta_1 \mapsto \ell_{\pi_1}, \ldots, \delta_n \mapsto \ell_{\pi_n}$ where the permutation π is such that $\ell_{\pi_1} \leq \ldots \ell_{\pi_n}$. This function is conceivable (although artificial) in HFC, but it is not implementable in NC, hence it is disallowed in HFC$'$. We remark however that all practically used built-in functions in HFC are definable with respect to those allowed by restriction R1.

Example 4.2 (About restriction R2). In order to show the rationale behind Restriction R2, consider the following HFC program

```
def hfc_avghood(x) { // : field(num) -> num
  foldhood(0, +, x) / foldhood(0, +, 1)
}
hfc_avghood(nbr{sns-temp()}) // : num
```

which on each device calculates the average temperature of neighbours. We may suppose that the same code, interpreted as an NC program, would calculate the same quantity. Instead, it is equivalent to the simpler NC program `sns-temp()`, which yields the temperature of the device where it is evaluated. If we evaluate the expression `nbr{sns-temp()}` against a neighbour δ', we obtain the temperature t' of that neighbour. Unfortunately, in the program the expression occurs outside of the scope of any `foldhood` construct and so it is evaluated against the device δ where it is evaluated. When function `hfc_avghood` is applied to t (the temperature of δ), the neighbour device δ' is ignored by both `foldhood` constructs, which fail to interpret the captured neighbouring value as such. The value of the program on device δ is then $n \cdot t/n = t$, where n is the number of neighbours of δ (including δ itself) and t is the value of the temperature on device δ.

We remark that an HFC program computing the average temperature of neighbours also when interpreted as an NC program can be conveniently written, by resorting to suitable programming patterns (illustrated in Section 4.6.2). In particular, for the example above, it is sufficient to make `x` a "by-name" parameter, thus obtaining the following HFC program:

```
def hfc_nc_avghood(y) { // : (() -> field(num)) -> num
  foldhood(0, +, y()) / foldhood(0, +, 1)
}
hfc_nc_avghood(() => {nbr{sns-temp()}}) // : num
```

We remark that all HFC programs considered in previous works [AVD+19, ABDV18, VAB+18] actually belong to HFC$'$ (or can straightforwardly be reformulated in order to do so). The following lemma provides a characterisation of HFC$'$ in terms of the type system in Figure 11.

Lemma 4.3 (Characterisation of HFC$'$). *The type system in Figure 11 is a restriction of the Hindley-Milner type system for HFC* [AVD+19] *enforcing restrictions R1 and R2.*

Proof. Restriction R1 is implicitly implemented in the definition of $\mathcal{D}_0$, in which the only built-in with field arguments is `map`. Restriction R2 is implemented in Rule [T'-FOLD], by requiring each free variable occurring in the third branch to be of local type – in the type system for HFC [AVD+19] `foldhood` is a built-in function, so there is no special rule for

`foldhood`-expressions. All the other rules are the same as for the Hindley-Milner type system for HFC [AVD+19]. □

4.4. **The NC′ Fragment of NC.** NC′ is the fragment of NC that can be typed by rules in Figure 11. These rules are significantly more restrictive then the rules in Figure 4, in particular:

- They acknowledge the existence of the four type categories, in particular of `field`(S);
- They demand that (anonymous and defined) functions have an allowed return type R;
- They require the sub-expressions of `rep`, `nbr` and `foldhood` to have local return type S;
- Finally, they require captured variables to have local type, in the body of both anonymous functions and folding expressions.

For convenience in the comparison with HFC′, we assume that NC′ comprises the `consthood` and `map` built-ins, even though in NC′ those functions could also be defined as follows.

```
def consthood(v) { v }
def map(f, v, ...) { f(v, ...) }
```

The embedding of NC′ as a fragment of NC can be formally characterised by means of the following definition and theorem.

Definition 4.4 (Erasure)**.** The *erasure* of an NC′ type T is the type **erasure**(T) obtained from T by replacing all occurrences of `field`(L) with L and dropping the distinction between the different kinds of type variables (i.e., considering each of them as a standard type variable t). Similarly, the erasure of a type scheme $\forall \overline{t}\overline{l}\overline{r}\overline{s}.L$ is the type scheme $\forall \overline{t}\overline{l}\overline{r}\overline{s}.\mathbf{erasure}(L)$ (dropping distinction between kinds of variables). Finally, the erasure of a type environment $\mathcal{A} = \overline{\mathbf{x}} : \overline{T}$ is **erasure**($\mathcal{A}$) = $\overline{\mathbf{x}}$: **erasure**($\overline{T}$); and the erasure of a type-scheme environment is **erasure**($\mathcal{D}$) = $\overline{\mathbf{x}}$: **erasure**($\overline{LS}$).

Theorem 4.5 (Typing Correspondence)**.** *Assume that* $\mathcal{D}; \mathcal{A} \vdash \mathbf{e} : T$ *in NC′. Then* **erasure**($\mathcal{D}$); **erasure**($\mathcal{A}$) $\vdash \mathbf{e} : $ **erasure**(T) *in NC.*

Proof. We proceed by induction on the syntax of $\mathbf{e}$. If $\mathbf{e} = \mathtt{nbr}\{\mathbf{e'}\}$, then $T = \mathtt{field}(S)$ where $\mathcal{D}; \mathcal{A} \vdash \mathbf{e'} : S$ by rule [T'-NBR]. By inductive hypothesis, **erasure**($\mathcal{D}$); **erasure**($\mathcal{A}$) $\vdash \mathbf{e'} :$ **erasure**(S) in NC. It follows by rule [T-NBR] that **erasure**($\mathcal{D}$); **erasure**($\mathcal{A}$) $\vdash \mathbf{e} :$ **erasure**(S), and the thesis follows since **erasure**(S) = **erasure**($\mathtt{field}(S)$).

In all other cases, the thesis follows since the NC rules can be obtained by removing restrictions from their counterparts in NC′: that types are restricted to peculiar kinds L, R or S (every rule except for [T'-VAR]); that captured variables have local type (in rules [T'-A-FUN] and [T'-FOLD]). □

4.5. **Equivalence between NC′ and HFC′.** In order to prove the equivalence between HFC′ and NC′, we first need to define what it means for an expression to have an equivalent behaviour in the two languages.

Definition 4.6 (Coherence)**.** Assume that $\mathcal{D}; \mathcal{A} \vdash \mathbf{e} : T$ in HFC′/NC′ where $\mathcal{A} = \overline{\mathbf{x}} : \overline{T}$. Let $\overline{\mathbf{v}}$ be HFC′ values of type $\overline{T}$, and let $\mathbf{v}(\delta)$ be defined by cases as $\ell(\delta) = \ell$, $\phi(\delta) = \ell$ where ℓ is such that $\delta \mapsto \ell \in \phi$.

We say that $\mathbf{e}[\overline{\mathbf{x}} := \overline{\mathbf{v}}]$ has the *same behaviour* in HFC′ and NC′ whenever:

```
def G(source, initial, metric, accumulate) {
  rep ( pair(source, initial) ) { (x) =>
    foldhood( pair(source, initial), min,
              pair(nbr{fst(x)}+metric(), accumulate(nbr{snd(x)})) )
} }
def T(initial, zero, decay) {
  rep ( initial ) { (x) => min(max(decay(x), zero), initial)
} }
```

FIGURE 12. The G and T blocks (code reported from literature [VAB+18]).

- if T is a local type, $\delta; \Theta; \sigma \vdash \mathsf{e}[\overline{\mathsf{x}} := \overline{\mathsf{v}}] \Downarrow \ell \langle \overline{\theta} \rangle$ if and only if $\delta, \delta; \Theta; \sigma \vdash \mathsf{e}[\overline{\mathsf{x}} := \overline{\mathsf{v}}(\delta)] \Downarrow \ell \langle \overline{\theta}' \rangle$ for some $\overline{\theta}$ and $\overline{\theta}'$;
- if T is a field type, $\delta; \Theta; \sigma \vdash \mathsf{e}[\overline{\mathsf{x}} := \overline{\mathsf{v}}] \Downarrow \phi \langle \overline{\theta} \rangle$ if and only if $\delta, \delta_i; \Theta; \sigma \vdash \mathsf{e}[\overline{\mathsf{x}} := \overline{\mathsf{v}}(\delta_i)] \Downarrow \ell_i \langle \overline{\theta}' \rangle$ for some $\overline{\theta}$ and $\overline{\theta}'$ where $\phi = \delta_i \mapsto \ell_i$.

Assuming that the above coherence condition holds for built-in functions, then it holds for every expression, as shown in the following theorem.

Theorem 4.7 (Equivalence between HFC′ and NC′). *Assume that for every built-in function b with local arguments, $\mathsf{b}(\overline{\mathsf{x}})$ has the same behaviour in HFC′ and NC′ for every substitution of $\overline{\mathsf{x}}$. Assume that $\mathcal{D}; \mathcal{A} \vdash \mathsf{e} : T$ in HFC′/NC′ where $\mathcal{A} = \overline{\mathsf{x}} : \overline{T}$, and let $\overline{\mathsf{v}}$ be HFC′ values of type $\overline{T}$. Then $\mathsf{e}[\overline{\mathsf{x}} := \overline{\mathsf{v}}]$ has the same behaviour in HFC′ and NC′.*

Proof. The proof follows from standard induction on expressions (see Appendix B.2). □

4.6. **NC Expressiveness.** In this section, we argue that NC is an expressive language for distributed computations. Section 4.6.1 shows that NC′ contains many relevant aggregate programs, including those providing universality for distributed computations. In Section 4.6.2, we argue that most of the HFC programs which are not directly interpretable in NC can be automatically refactored (while preserving their behaviour) in order to fit within HFC′ (hence NC′), enlarging the class of aggregate programs expressible in NC to virtually all HFC programs. This process is exemplified in Section 4.6.3, where the self-stabilising fragment of HFC [VAB+18] is ported to NC through the refactoring just introduced. Finally, Section 4.6.4 argues that the NC programs that are not in NC′ can fruitfully extend the expressive power of HFC.

4.6.1. *NC′ examples: G and T blocks and universality.* The correspondence between NC′ and HFC′ given by Theorem 4.7, although restricted to a fragment of the two languages, is applicable to many relevant aggregate programs, allowing for the automatic transfer of algorithms and properties proven in HFC to NC. As a first paradigmatic example, consider the G and T blocks, proposed as part of a combinator basis able to express most aggregate systems [BV14]. The HFC formalisation of their code [VAB+18] is reported in Figure 12, and has the same behaviour in NC since it actually belongs to HFC′/NC′.

As a further example, consider the code in Figure 13, which is also in HFC′/NC′ and encodes the behaviour of any distributed Turing Machine (expressed as a function f taking as input in every firing the whole collection of causally available data) as an HFC function.

```
// previous round value of v
def older(v, null) {
  fst(rep (pair(null, null)) { (old) => pair(snd(old), v) })
}
// gathers values from causal past events into a labelled DAG
def gather(node, dag) {
  let old = older(dag, dag_empty()) in
  let next = dag_join(foldhood(old, dag_union, nbr{dag}), node) in
  if (next == node) { dag } { gather(node, dag_union(dag, next)) }
}
def f_field(e, v...) {
  f( gather(dag_node(e, v...), dag_node(e, v...)) )
}
```

FIGURE 13. The universal translation of distributed Turing Machines (code reported from literature [ABDV18]).

This code was used to prove *Turing Universality* for HFC,[12] but in fact it also proves that HFC$'$/NC$'$ hence NC are Turing-universal as well (assuming a sufficient collection of built-ins).

4.6.2. *Refactoring of HFC programs into HFC$'$.* Despite many common aggregate functions being in HFC$'$/NC$'$, not every relevant such function belongs to this fragment. In particular, the restrictions imposed by the type system in Figure 11 prohibit the common pattern of functions with field arguments, folding those arguments in their body. However, refactoring strategies exist that we can use in order to turn most HFC programs into HFC$'$ while preserving their behaviour, by converting a field argument of a function into a local argument, thus allowing its capture within `foldhood` statements.

(1) *Abstracting*: a field argument of a function may be passed "by name" through

$$((\mathtt{x})\texttt{=>}\{\mathtt{e_1}\})(\mathtt{e_2}) \quad \longrightarrow \quad ((\mathtt{x})\texttt{=>}\{\mathtt{e_1}[\mathtt{x} := \mathtt{x}()]\})(()\texttt{=>}\{\mathtt{e_2}\})$$

This refactoring preserves the behaviour of a program, provided that either *(i)* the abstracted parameter does not depend on the current domain (e.g., relational sensors such as `nbrRange`), or *(ii)* the parameter does not occur within branches in the body of the function (so that the evaluation is performed in the function with respect to the same domain as in the argument). If instead it does occur within a branch and depends on the current domain, its deferred evaluation within the function body will generally produce a different result. For example, consider function `counthood` counting the number of neighbours in the current domain:

```
def counthood() { foldhood(0, +, 1) }
```

and suppose that the argument is $\mathtt{e_2} = \mathtt{nbr}\{\mathtt{counthood()}\}$, and the function $\mathtt{e_1}$ is

$$\mathtt{if}(\mathtt{temperature}() < 30)\{0\}\{\mathtt{foldhood}(\mathtt{counthood}(), \max, \mathtt{x})\}.$$

This functions returns 0 on *"cold"* devices; otherwise, it returns the maximum number of <u>total</u> neighbours that any of the *hot* neighbours has (a metric that may be used, e.g.,

---

[12]A programming model for distributed systems is Turing-universal if and only if it is able to replicate the behaviour of any distributed Turing machine.

to trigger an alarm). After the *abstracting* refactoring, this result changes slightly to the maximum number of <u>hot</u> neighbours that any of the *hot* neighbours has. This reduces the computed value, possibly reducing the effectiveness of the metric and delaying the triggering of the alarm. However, most programs with this characteristic can still be refactored, by resorting to the following extended refactoring whenever the domain dependency is due to sub-expressions of local type.

(2) *Abstracting with parameters*: a field argument $\mathsf{e}_2[\overline{\mathsf{e}}']$ where $\overline{\mathsf{e}}'$ have local type can be passed as a function with parameters through

$$((\mathsf{x})\texttt{=>}\{\mathsf{e}_1\})(\mathsf{e}_2[\overline{\mathsf{e}}']) \quad \longrightarrow \quad ((\mathsf{x},\overline{\mathsf{y}})\texttt{=>}\{\mathsf{e}_1[\mathsf{x}:=\mathsf{x}(\overline{\mathsf{y}})]\})((\overline{\mathsf{y}})\texttt{=>}\{\mathsf{e}_2[\overline{\mathsf{y}}]\},\overline{\mathsf{e}}')$$

This refactoring can address the previous example, by leaving the `counthood()` sub-expression (which is the one depending on the current domain and has local type) as a parameter:

$$((\mathsf{x},\mathsf{y})\texttt{=>}\{\texttt{if}(\texttt{temperature}() < 30)\{0\}\{\texttt{foldhood}(\texttt{counthood}(),\max,\mathsf{x}(\mathsf{y}))\}\})$$
$$((\mathsf{y})\texttt{=>}\{\texttt{nbr}\{\mathsf{y}\}\},\texttt{counthood}())$$

Theorem 4.8 (Correctness of the abstracting with parameters refactoring). *Assume an HFC expression* e *consists in a function* $(\mathsf{x})\texttt{=>}\{\mathsf{e}_1\}$ *called with a field argument* $\mathsf{e}_2[\overline{\mathsf{e}}']$ *where* $\overline{\mathsf{e}}'$ *all have local type, and consider the refactored expression* e'' *as in (2). If the argument* x *does not occur within branches in* e_1*, then* e'' *has the same behaviour as* e*. If the result of* $\mathsf{e}_2[\overline{\mathsf{y}}]$*, for any given values for* $\overline{\mathsf{y}}$ *and for any domain of computation* $\mathbf{dom}(\Theta)$*, is a field value* ϕ *such that* $\phi(\delta)$ *does not depend on* $\mathbf{dom}(\Theta)$ *for any* δ*, then* e'' *has the same behaviour as* e*.*

Proof. First, notice that the semantics of HFC is compositional except for the set of aligned neighbours: in other words, the evaluation result of an expression depends on the enclosing expression only in the determination of the set of aligned neighbours; once that set is fixed, the result of the expression is fixed too. The effect of refactoring (2) is shifting the evaluation of $\mathsf{e}_2[\overline{\mathsf{e}}']$ in a different part of the evaluation tree: function call itself (by Rule [E-APP]) to several points in the function body (the occurrences of x).

If the evaluation of $\mathsf{e}_2[\overline{\mathsf{e}}']$ does not depend on the domain of evaluation, then this computation shift has no effect, concluding the proof for that case. If x does not occur within branches, the set of aligned neighbours for each of its occurrences is the same as the set of aligned neighbours for the function call itself. It follows that the evaluation of $\mathsf{e}_2[\overline{\mathsf{e}}']$ in place of every occurrence of x has to produce the same result as in the argument, concluding the proof. $\square$

The following simpler refactoring can address cases when the argument is a simple `nbr`-expression, regardless of branches in e_1.

(3) *Deferring*: an `nbr` in the argument may be transferred into the body as

$$((\mathsf{x}) \texttt{=> } \{\mathsf{e}_1\})(\texttt{nbr}\{\mathsf{e}_2\}) \quad \longrightarrow \quad ((\mathsf{x}) \texttt{=> } \{\mathsf{e}_1[\mathsf{x}:=\texttt{nbr}\{\mathsf{x}\}]\})(\mathsf{e}_2)$$

This refactoring is a simplification of refactoring (2), abstracting $\texttt{nbr}\{\mathsf{e}_2\}$ with respect to the parameter e_2 of local type. In fact, refactoring (2) in this case would produce:

$$((\mathsf{x},\mathsf{y})\texttt{=>}\{\mathsf{e}_1[\mathsf{x}:=\mathsf{x}(\mathsf{y})]\})((\mathsf{y})\texttt{=>}\{\texttt{nbr}\{\mathsf{y}\}\},\mathsf{e}_2)$$

and inlining $(\mathsf{y})\texttt{=>}\{\texttt{nbr}\{\mathsf{y}\}\}$ into the occurrences of the first parameter x yields exactly the result of refactoring (3).

Notice that the general refactoring (2) and its common special cases (1) and (3) cover practically any realistic situation. In order for neither of those to be applicable, we would need the argument e_2 to be a field expression which depends on the current domain (and not only because of sub-expressions of local type), and which is passed into a function e_1 folding it inside a branching statement: we are not aware of any meaningful or realistic situation meeting these requirements. Thus, through these refactorings we can convert practically any HFC program into HFC′, hence into NC, although the resulting refactoring may be extensive: since the refactoring of a function depends on its call patterns, more than one version of a function may be needed to cover all of them. However, this is *not* an issue for a "native" NC programmer, which would just interpret the above refactorings as *programming patterns*, following them from the start of application development. Interpreted as programming patterns, the above rules suggest deferring `nbr` applications when possible, or using call-by-name instead of call-by-value on field arguments otherwise.

4.6.3. *An example: refactoring the self-stabilising fragment of HFC.* A paradigmatic example of HFC programs that can be converted into HFC′ through the refactorings in Section 4.6.2 is the *self-stabilising fragment* [VAB+18]. We say that a time- and space-distributed data is *stabilising* iff it remains constant in every point after a certain time t_0, and its *limit* is the value assumed after t_0. We say that a distributed program is *self-stabilising* iff given stabilising inputs and topology, it produces a stabilising output which depends only on the limits of the inputs and topology (and not on the concrete scheduling of events, nor on the input values before stabilisation).

A subset of HFC, called *self-stabilising fragment*, has been proved in the literature [VAB+18] to consist of self-stabilising programs only. This fragment makes use of functions folding field arguments, and thus is *not* part of HFC′ and cannot be directly translated into NC preserving its behaviour (or self-stabilisation). However, the refactorings in Section 4.6.2 can be applied to obtain the equivalent fragment in Figure 14 that does belong to HFC′/NC′. The self-stabilising fragment combines syntactic requirements with mathematical requirements on functions, annotated in the figure through superscripts on function names.

C (**Converging**). A function $f(v_1, v_2, \overline{v})$ is said converging iff, in every firing, its return value is closer to v_2 than the maximal distance that the two arguments v_1 and v_2 have in any neighbour firing (according to any metric measuring that distance).

M (**Monotonic non-decreasing**). A stateless[13] function $f(x, \overline{x})$ with arguments of local type is monotonic non-decreasing in its first argument iff whenever $\ell_1 \leq \ell_2$, also $f(\ell_1, \overline{\ell}) \leq f(\ell_2, \overline{\ell})$.

P (**Progressive**). A stateless function $f(x, \overline{x})$ with local arguments is progressive in its first argument iff $f(\ell, \overline{\ell}) > \ell$ or $f(\ell, \overline{\ell}) = \top$ (where $\top$ is the maximal element of the relevant type).

R (**Raising**). A function $f(\ell_1, \ell_2, \overline{v})$ is raising with respect to partial orders $<, \lhd$ iff: *(i)* $f(\ell, \ell, \overline{v}) = \ell$; *(ii)* $f(\ell_1, \ell_2, \overline{v}) \geq \min(\ell_1, \ell_2)$; *(iii)* either $f(\ell_1, \ell_2, \overline{v}) \rhd \ell_2$ or $f(\ell_1, \ell_2, \overline{v}) = \ell_1$.

---

[13]A function $f(\overline{x})$ is *stateless* iff given fixed inputs $\overline{v}$ always produces the same output, independently of the environment or specific firing event. In other words, its behaviour corresponds to that of a mathematical function.

$$\begin{array}{rcl} \texttt{s} & ::= & \texttt{x} \mid \texttt{v} \mid \texttt{let x} = \texttt{s in s} \mid \texttt{f}(\overline{\texttt{s}}) \mid \texttt{if(s)\{s\}\{s\}} \mid \texttt{nbr\{s\}} \quad \text{self-stabilising expression} \\ & & \mid \texttt{rep(e)\{(x)=> \{f}^{\texttt{C}}\texttt{(x, s, \overline{e})\}\}} \\ & & \mid \texttt{rep(e)\{(x)=> \{f((y,z)=> \{mux(nbr\{y\} < y, nbr\{x\}, z)\}, \overline{s})\}\}} \\ & & \mid \texttt{rep(e)\{(x)=> \{f}^{\texttt{R}}\texttt{(foldhood(s, min, f}^{\texttt{MP}}\texttt{(nbr\{x\}, \overline{s})), x, \overline{e})\}\}} \end{array}$$

FIGURE 14. Syntax of a self-stabilising fragment of field calculus expressions, where self-stabilising expressions `s` occurring inside a `rep` statement cannot contain free occurrences of the `rep`-bound variable `x`.

Theorem 4.9 (Self-stabilising fragment of NC). *Every self-stabilising expression according to the fragment in Figure 14 is a self-stabilising NC expression.*

Proof. Notice that the fragment in Figure 14 can be obtained from that in [VAB$^+$18, Figure 2] by means of the following two refactorings.

- In the *converging* `rep` expression $\texttt{rep(e)\{(x)=> \{f}^{\texttt{C}}\texttt{(x, s, \overline{e})\}\}}$, deferring is applied to transfer `nbr` in arguments to the body of the function $\texttt{f}^{\texttt{C}}$, adapting the definition of *converging* function accordingly. By Theorem 4.8, these expressions have the same behaviour as those in [VAB$^+$18, Figure 2], which are self-stabilising by [VAB$^+$18, Theorem 1].
- In the *acyclic* `rep` expression $\texttt{rep(e)\{(x)=> \{f((y,z)=> \{mux(nbr\{y\} < y, nbr\{x\}, z)\}, \overline{s})\}\}}$, abstraction is applied to the first argument with respect to parameters $\texttt{y}, \texttt{z}$ (whose values are added into the additional parameters $\overline{\texttt{s}}$ passed to `f`). By Theorem 4.8, these expressions have the same behaviour as those in [VAB$^+$18, Figure 2], which are self-stabilising by [VAB$^+$18, Theorem 1].

The rest of the fragment is identical modulo inessential modifications (expansion of the `nbrlt` and `minHoodLoc` functions into their definition), concluding the proof. □

4.6.4. *NC programs that cannot be straightforwardly expressed as HFC programs.* As argued in [ADVC16], programs such as *updatable metrics* and *combined Boolean restriction* are not conveniently expressed in HFC. In the former case, we can use the following general scheme for updatable functions, first proposed in [AVD$^+$19]:

```
def up(injecter) {
  snd( rep(injecter()) {
    (x) => { foldhood(injecter(), max, nbr{x}) }
} ) }
```

where `injecter` is a function returning a pair ⟨`version number`, `function code`⟩, and the built-in operator `max` selects the pair with the highest version number among its arguments. This procedure defines a perfectly reasonable "upgradeable function" by spreading functions with higher version number throughout devices. However, it is not allowed by the type system of HFC for functions returning fields, such as metrics (which usually have type $() \to \texttt{field(num)}$). This scheme can instead be used in NC (as shown in Section 6), and works properly provided that new versions are injected at a slow rate, and an occasionally empty domain of a field-like expression does not produce critical effects.

Another situation where the permissive behaviour of NC is crucial is that of *combined Boolean restriction*. In this setting, a field-like expression `e` needs to be restricted to those devices agreeing on the value of n Boolean parameters $\texttt{b}_1, \ldots, \texttt{b}_n$, before being folded with

foldhood(v, f, e). This rather abstract example might be concretely instantiated, e.g., in case an aggregation needs to be executed separately on devices with different configurations. In HFC, this effect can be achieved only by restricting on each of the 2^n possibilities for the parameters, as in the following.

```
if (b1 && b2 && ... bn) {foldhood(v, f, e)} {
  if (!b1 && b2 && ... bn) {foldhood(v, f, e)} {
    if (b1 && !b2 && ... bn) {foldhood(v, f, e)} { ... }}}
```

However, such a program might be infeasibly large even for small values of n. On the other hand, in NC the above program can be concisely rewritten as:

```
foldhood(v, f, e + if (b1) {nbr{0}} {nbr{0}} + if (b2) {nbr{0}} {nbr{0}} + ...)
```

whose size is linear in n.[14] The domain of the i-th 0-valued field-like subexpression above is equal to the set of devices agreeing on b_i, hence by intersecting all of them the resulting domain corresponds to the set of devices agreeing on each of the n given parameters.

5. ScaFi: an Implementation of the neighbours calculus

In this section, we present ScaFi, an implementation of NC embedded in the Scala programming language.

While *external* (or *standalone*) domain-specific languages (DSLs) have their own syntax and semantics, *internal* DSLs are *embedded* into some other language [Voe13]. Developing an internal DSL brings major benefits in terms of *reuse* of features, toolchain support, and familiarity with the host language—at the expense of the syntactical and semantical constraints imposed by the host. Scala has been chosen as the host language of ScaFi for its modernity, its feature set (it provides mechanisms supporting the creation of expressive APIs and DSLs [AHKY15]) and its ability to target and reuse libraries from various execution platforms (such as the JVM, but also Javascript through the ScalaJS project [Doe18]). With respect to other field calculi, NC lends itself to smooth implementation in Scala, as e.g. there are no neighbouring fields to be dealt with at the typing, syntactical, and semantical level. A Scala expression of type `Int` is automatically interpreted in ScaFi as an aggregate program producing a field of `Int`, completely fulfilling the "everything is a field" view.

5.1. The Scala implementation of NC.
The following interface, implemented as a Scala trait, represents the basic NC constructs as methods:

```scala
trait Constructs {
  // Key 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 aggregate[A](b: => A): A

  // Abstract types
  type ID;        // type of device identifiers
  type LSNS, NSNS; // type of local and neighbour sensor names
```

---

[14]In this code we assumed that `x` has numerical type, but similar code can be obtained for any type by defining a binary operator which is the identity on its first argument.

```scala
  // Contextual, but foundational
  def mid(): ID
  def sense[A](name: LSNS): A
  def nbrvar[A](name: NSNS): A
}
```

In Scala, methods are introduced with the def keyword, can be generic (with type parameters specified in square brackets), may accept multiple parameter lists, and specify a return type at the end of the signature (when this is not given the compiler attempts inference). Function types may take the form $(I_1,\dots,I_N)$=>O, which is actually syntactic sugar over FunctionN$[I_1,\dots,I_N,O]$; curried function types can be written as I_1=>$\cdots$=>I_N=>O (=> is right associative). Tuple types may take the form $(T_1,\dots,T_N)$, which is actually syntactic sugar over TupleN$[T_1,\dots,T_N]$; similarly, a literal tuple value can be denoted as $(v_1,\dots,v_N)$. By-name parameters, denoted with type =>T, capture expressions or blocks of code that are passed unevaluated to the method and are actually evaluated every time the parameter is used—they are basically syntactic sugar over 0-ary function types. As a relevant note on syntax, especially useful in DSLs to render constructs with code blocks, unary parameter lists in a method can be called also with curly brackets instead of parentheses. E.g., all the following are valid invocations for rep method above: rep$(\cdot)(\cdot)$, rep$(\cdot)\{\cdot\}$, rep$\{\cdot\}(\cdot)$, rep$\{\cdot\}\{\cdot\}$. Finally, nullary methods can be invoked without parentheses; e.g., mid is a valid method call just like mid().

First of all, notice that the input and output types of the constructs (especially nbr) are proper, i.e., no field-like datatype appears in the signatures. Then, compare trait Construct with the NC syntax from Section 3.1. Beside the curried form of rep and foldhood, the only significant difference is an additional construct aggregate which is used to turn standard Scala functions/methods into field calculus functions, i.e., as units of alignment (refer to Section 3.3.1 for details). We report here the encoding of if, which is called branch in ScaFi due to the fact that if is a reserved keyword in Scala:

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

Namely, the then and else expressions are passed unevaluated (as by-named arguments) to branch, and there used in the bodies of two corresponding lambdas, wrapped with aggregate (which could be seen as a form of tagging). The mux on cond is used to select one of those functions, which is then invoked straight on via operator ().

Moreover, it is crucial to note that the programming patterns discussed in Section 4.6.2 (abstraction and deferring) are supported in a syntactically transparent way thanks to the Scala support of by-name parameters. Indeed, the refactoring shown in Section 4.6.2 are automatically applied.

Finally, in practice, an aggregate program can be defined in ScaFi by extending class AggregateProgram and implementing a method called main which defines the main program expression. This class provides an implementation of the Constructs trait: by subclassing it, the syntax and semantics of the ScaFi DSL is made available within the subclass definition, such that its objects could be used both for simulation purposes or to encapsulate the aggregate logic in actual distributed implementations.

```scala
class GradientProgram extends AggregateProgram
```

```
  with Gradients          // brings gradient() in
  with StandardSensors   // brings nbrRange() in
{
  // Custom definitions
  def isSource = sense[Boolean]("source")

  // Main expression
  def main(): (Int,Double) = {
    // Return a 2-element tuple (round count, gradient value)
    (rep(0)(r=>r+1), gradient(isSource, nbrRange))
  }
}
```

A full description of ScaFi is beyond the scope of this paper: the interested reader can refer to [CV18] for more details.

5.2. **On pragmatics: neighbouring fields vs computation against a neighbour.** In this section, we provide a hint to the practical issues that NC/ScaFi solve, enabling easy implementations *and* a clean support for aggregate programming.

Consider the ScaFi expression of the classic gradient algorithm (cf. Example 3.10):

```
rep(Double.PositiveInfinity) { case d =>
  mux(source) { 0.0 } { minHood(nbr(d) + metric()) }
}
```

and focus on expression `nbr(d)+metric()`. While in NC/ScaFi the latter is a sum of two local values, computed against aligned neighbours (by the `*hood` operator), in the field calculus it would be a sum of two neighbouring fields. While in a standalone DSL, implementations may transparently lift local operators to work with fields (i.e., collection-like objects), this would not typically be easy or even possible in internal DSLs. Suppose a neighbouring field of `T`-typed objects is represented as an instance of type `Field[T]`. In a purely functional world, two fields could be combined with a "classical" `map2` function:

```
map2(nbr(d), metric(), _+_)
```

However, this would introduce some clutter in aggregate programs and require a versatile set of functional combinators. In a powerful object-oriented language with mechanisms like extension methods (such as Scala or C#), it would be possibly to manually add specific operators to work with fields.

```
implicit class NumericField[T:Numeric](f: Field[T]){
  def +(f2: Field[T]): Field[T] = map2(f, f2, implicitly[Numeric[T]].plus(_,_))
  // ...
}
// Assuming nbr(d) has type Field[Double]
nbr(d) + metric // possible by Scala's implicits (cf. [OMO10]) + operator syntax
```

This example leverages Scala's *pimp-my-library idiom* [OMO10], which uses implicits to enable automatic, compile-time conversion of a `Field` of `Numeric`s to a `NumericField` that could accept a method `+`. However, the lifting of local operators to fields would be manual, and not automatic—introducing a overhead on library development. Some language could provide advanced features to mitigate this issue (e.g., through powerful compile-time macro

mechanisms), but in this case we would pose severe constraints on the host language for the internal DSL, hence limiting the development environment where aggregate programming could be supported.

6. Case Study

The goal of this section is to show how NC/ScaFi can be used to eloquently express collective behaviour. More specifically, the goal is not to show that NC/ScaFi necessarily provide a better programming support with respect to e.g. HFC/Protelis—though the Scala embedding does provide various practical benefits, through reuse of Scala's powerful type system and features. Indeed, the benefit of NC lies in a conceptual frugality (by avoiding the "neighbouring field" notion) that leads to a different computational model (cf., Section 4) with easier DSL embeddability in mainstream programming languages (cf. Section 5). Therefore, this section shows that such a model and implementation can be used to smoothly program increasingly complex field coordination-based systems. In particular, we apply the programming model to the context of edge computing [SCZ+16] and ad-hoc cloudlets [CHL+15] (introduced in Section 6.1). Evaluation of correctness of the proposed solution is performed by simulation. The source code, configuration files, and instructions for launching the experiments and reproducing the presented results is publicly available at an online and permanent repository[15] [Cas22b].

6.1. **Ad-hoc Edge Computing Support.** Edge computing [SCZ+16] is a recent paradigm that aims to bring cloud-like functionality (e.g., virtualisation, elastic provisioning of resources, and "anything-as-a-service") closer to end devices at the edge of the network. Its goal is to ultimately improve the Quality of Service (QoS) of situated systems as in the IoT, e.g., through reduction of communication latencies, bandwidth and energy consumption, essentially achieved by shortening the round-trip path of data to/from elastic resource pools.

In this case study for ScaFi, we focus on *ad-hoc* edge computing—i.e., a decentralised form of edge computing that does not rely on pre-existing infrastructure but is rather supported by a large-scale network of devices logically interacting in a peer-to-peer fashion. This may also be a first step for merging volunteer computing [DS14] and edge computing into a form of *volunteer edge computing* where devices with spare resources make them available (possibly by incentives) to nearby users.

We assume to operate in a region of a smart city where hundreds of devices willing to offer resources run the Aggregate Computing middleware and may hence participate in the Aggregate Edge Computing (AEC) application. In a vision of smart city operating systems, it is indeed sensible to assume that devices willing to fully benefit from smart city services are asked to give their "social contribution" by participating in the system as well, where, of course, we also assume proper security and privacy systems are in place. The system is *open*, in the sense that devices may enter or leave the system as they like. For this paper we abstract from the way in which tasks are assigned to devices.

The goal of AEC is to support a *self-organisation process* to monitor resource availability and usage (i.e., load) in the system, and spread information as well as directives which can be exploited by devices for decentralised control activities and by users for determining advantageous deployment options. The idea is to provide support for edge computing

---

[15]https://github.com/metaphori/experiments-neighbours-calculus.

by leveraging the space-time distribution of resources. Managing edge resources in a collective, space-time-oriented fashion is meant to provide the following benefits: *(i)* it makes straightforward to exploit the locality principles that often sustain IoT, CPS, and smart city applications, *(ii)* it allows us to assume that devices are only able to communicate with other nearby devices through short-range wireless technologies (e.g., 10 to 100 meters range); *(iii)* it allows us to relate the spatio-temporal distribution of resources with human social activity; and *(iv)* it provides a natural way to handle collective metrics and take into account contextual data and constraints at multiple scales.

6.2. **Model and Design.** We model devices of the AEC system as self-aware, situated entities that can sense the resources they own (e.g., number of CPUs, memory, and storage) and their current levels of utilisation.

```scala
def getLocalProcessing() = sense[ProcessingStat]("processing")
def getLocalStorage() = sense[StorageStat]("storage")
// ...
def getLocalResources() = LocalStats(getLocalProcessing, getLocalStorage, ...)
```

Data can be easily modelled in Scala with case classes (i.e., algebraic data types with built-in equality, pattern matching support, etc.).

```scala
// Just for simplcity of exposition, CPUs are not shared.
case class ProcessingStat(numCPUs: Int = 0, mHz: Int = 0, numCPUsInUse: Int = 0)

trait DiskType; case object HDD extends DiskType; case object SSD extends DiskType
case class StorageStat(MiBs: Int = 0, dtype: DiskType = HDD, MiBsInUse: Int = 0)

case class LocalStats(ps: ProcessingStat, ss: StorageStat, ...)
```

We also assume the devices can communicate with other devices in their vicinity—i.e., the neighbouring relationship is basically spatial.

To divide the complexity of the resource monitoring task, we split the whole environment into manageable *areas* (which may also be called *cloudlets*) that can be monitored and controlled more easily. In order to solve issues related to distributed consensus, we want the system to resiliently self-organise to elect a leader in each area, so as to provide consistent views of the state of areas as well as enacting global decisions for an area. That is, the leaders effectively act as decentralised management points which are responsible for orchestrating a subset of devices in the system—so, in a sense, we apply a hybrid approach that combines orchestration and choreography as suggested in [VAA+18]. In practice, leaders act as sinks of contributions from the devices of the corresponding area, and as propagators of area-wide context information back to its feeders; but, in order to implement such a process, we need mechanisms for resiliently streaming information across dynamic space-time regions. The following section shows library functions that can be used to streamline the implementation of aforementioned mechanisms.

6.3. **Building Blocks.** Implementing the above ideas requires a careful crafting of the aggregate specification. There is need of identifying some recurrent patterns (e.g., peer-to-peer propagation of information in- and out-ward an area) that may come handy in many situations. They can then be used to raise the abstraction level through a reusable API

that hides low-level detail (e.g. the interplay of `rep` and `nbr`) only exposing the functional contract expressing the relationship between input fields and output fields. In doing so, we progressively move from a device-centric view (i.e., local sensing and neighbour-oriented communication) to an aggregate view (i.e., collective behaviour and data) of the system, regaining declarativity and intent.

In this case study, we leverage the following building blocks, which are available in the ScaFi standard library.

```
case class Gradient(algorithm: (Boolean,()=>Double)=>Double,
                    source: Boolean, metric: () => Double)
def G[V](gradient: Gradient, field: V, acc: V => V): V
def C[P: Bounded, V](potential: P, acc: (V, V) => V, local: V, Null: V): V
def S(grain: Double, metric: () => Double): Boolean
```

Their implementation is already described in other papers [BPV15, VAB+18] so here we present them by a functional perspective and just provide some insight about how they could work internally.

Recall the notion of gradient from Example 3.10: a field computation that, from a Boolean field denoting a region, returns the field of minimum distances from that region. The gradient field is key in spatial computations because it provides a direction from any point to a target: by descending (or ascending, resp.) the gradient field, i.e., by following the minimum (or maximum, resp.) values locally observed, one moves close (or away, resp.) to a certain target. With case class `Gradient`, we package together the algorithm and input fields—e.g., it may be useful to ensure that both a particular gradient and another function using it adopt the same metric. The *gradient-cast* operator, also known as `G`, is a generalised form of the gradient computation that accumulates values through a function `acc` along a gradient field, starting with the given `field` at sources and new devices. So, functionally `G` maps a source field and a field of values to a field whose values are progressively transformed, distance-wise, from source values. For instance, a gradient field can be constructed via `G` as follows.

```
val src = sense[Boolean]("source")
val classicGradient = Gradient(gradient(_,_), src, nbrRange)
G[Double](classicGradient, mux(src){ 0 }{ inf }, _+classicGradient.metric())
```

With `G`, it is also trivial to define a `broadcast` function for propagating information outward from a source, by ascending the potential field.

```
def broadcast[V](source: Boolean, field: V): V =
  G[V](classicGradient, field, (v: V) => v)
```

A change of `field` where `source` is true basically generates a new wave in this continuous streaming of information. Notice that `field` has a value everywhere but only the value of the `source` is selected and then identically preserved in the propagation. Also, notice that both `G` and `broadcast` may admit a different signature where the "potential field" is given as input and not calculated internally.

Dual to the gradient-cast operator is the *converge-cast* operator, also known as `C` [ACD+21]: indeed, the conceptually inverse notion of a propagation from a point outward is a propagation inward, from the edge towards a point. Functionally, `C` accumulates a `local` field along some `potential` field (yielding `Null` if no direction to descend can be found), where the final results of the accumulation end up collected at points of zero potential. In the

signature above, we show that we may abstract from the concrete type `P` of the potential field (which is usually a `Double`), provided that there exists an ordering over `P` values—as enforced by the context bound constraint `P:Bounded` (which is the mechanism for the typeclass idiom in Scala [OMO10], where the method may be called provided an implicit instance of type `Bounded[P]` can be statically resolved). It is also worth pointing out that both `G` and `C` are self-stabilising operators, whose compositions are also self-stabilising [VAB+18] (see the discussion in Section 4.6.1); hence, there are formal guarantees that programs built using only self-stabilising constructs like `G` and `C` will eventually reach a fixpoint once inputs stop changing.

The last fundamental building block that we cover is the *sparse-choice operator*, also known as `S`, which yields a Boolean field holding `true` in correspondence of elected nodes (also called *leaders*) which are selected in a manner such that adjacent leaders are at about distance `grain`. In practice, it performs a leader election process which also results in a split of the space into areas of a diameter which is approximately `grain`, since any connected device will be at a distance at most `grain` from a leader (think of a gradient from the leader field). Internally, an implementation of `S` may work by performing a distance competition among devices: initially, every device will propose itself as a leader, but initially assigned random values (ultimately discriminated by device IDs in case of breakeven) will be used to break symmetry (e.g., by selecting the minimum); if the gradient from the currently selected leader is larger than `grain` at a device, then it will start another competition for leadership of another area.

6.3.1. *Upgradeable functions.* The case study also uses the notion of *upgradeable function* explained in Section 4.6.4. The idea is to support hot upgrades of application algorithms by allowing injection of versioned functions at some device of the system and then spreading such novel versions through a gossiping process that eventually settles everywhere the function of the higher version. Concretely, we model versioned functions and injectors as types of the form

```
case class Fun[R](ver: Int, fun: ()=>R)
type Injecter[R] = () => Fun[R]
```

and define a function `up`,

```
def up[R](injecter: Injecter[R]): Fun[R] = rep(injecter()){ f =>
  foldhood(injecter())((f1,f2) => if(f1.ver>f2.ver) f1 else f2)(nbr{f})
}
```

to handle the upgrade process through corresponding propagation and selection of functions.

6.4. **Implementation.** Considering the modelling and library functions covered in previous sections, the core specification of the case study can be expressed as in the following ScaFi program.

```
class AdhocCloud extends AggregateProgram
  with StandardSensors with BlockG with BlockC with FieldUtils {

  val grain = 150 // parameter: mean distance between leaders

  def main = {
```

```
    // Get a metric for S, dynamically
    var Smetric = up[Double](metricInjecter(k)).fun
    // Resilient leader election
    val leaders = S(grain, Smetric)
    // Use the standard neighbouring range metric for the other operations
    val metric = nbrRange
    // Potential field to leaders
    val potential = gradient(leaders, metric)
    // Query local resources
    val localResources = getLocalResources()
    // Collect resources information towards leaders
    val resourcesPerArea =
      C[Double,Stats](potential, _++_, Map(mid->localResources), Map())
    // Broadcast from leaders outwards aggregated statistics
    val allResourcesInArea = broadcast(potential, metric,
      resourcesPerArea.values.foldLeft(AggregateStats())(_.aggregation(_)))
    // ...
  }
}
```

The flow is simple and reasoning is simplified by the collective stance and compositionality of Aggregate Computing: we use S to split the system into cloudlets monitored by corresponding leaders (where the metric used to create the areas can be dynamically updated); we build a gradient from leaders to set the pathways for collecting data into leaders and propagating the determinations of leaders to their workers; and finally, we use those pathways to stream the cloudlet-wise estimation of resource availability and usage to all the members. By the way, notice that this resource monitoring example represents the application of a more general pattern [CPVN19] that finds application in other large-scale, distributed coordination scenarios such as for situated problem solving [CTVD19] and client/server task allocation [CV19].

6.5. **Evaluation: Correctness.** The system has been simulated in the Alchemist simulator [PMV13, VCP16b]—a graphical snapshot is given by Figure 15[16], whereas Figure 16 provides empirical evidence of its correctness. The simulated system consists of 400 devices, each with a random amount of resources, dispersed unevenly around the downtown of Cesena, Italy. Mobility is not considered. A certain level of load is simulated in the system, and we perturb it with spikes of high load. The goal of the system is to reactively adjust the estimate of the load. As for additional perturbations, we also introduce a probability for temporary failure of leaders. Finally, in order to show the positive impact of updatable metrics, we inject a new metric to fine tune the dimension of areas at runtime—other aspects of configuration can be examined in the repository.

7. Related Work

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

---

[16]Full-size colour pictures are available at the provided repository.

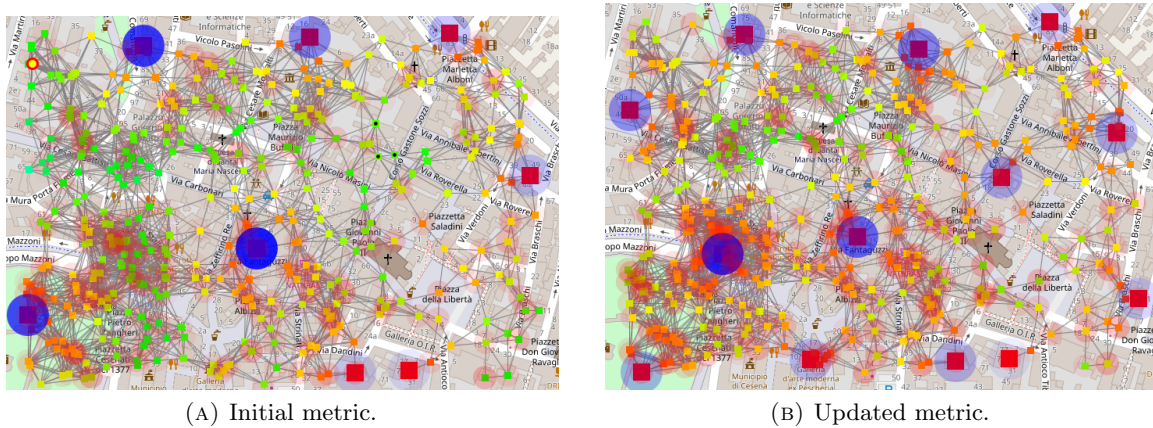(A) Initial metric.

(B) Updated metric.

FIGURE 15. A visual representation of the scenario, with two snapshots taken before and after a hot update of the metric. The big red squares represent leader nodes. Superimposed with any leader is a semi-transparent blue ball whose intensity is proportional to the amount of resources available in the corresponding area. Other nodes have a colour representing the gradient field towards leaders (warmer, red-like colours for nodes close to the leader, and colder, green-like colours for peripheral nodes). The updated metric in (b) provides an overestimation of distances which results in smaller areas or, equivalently, in more cloudlets (and leaders).
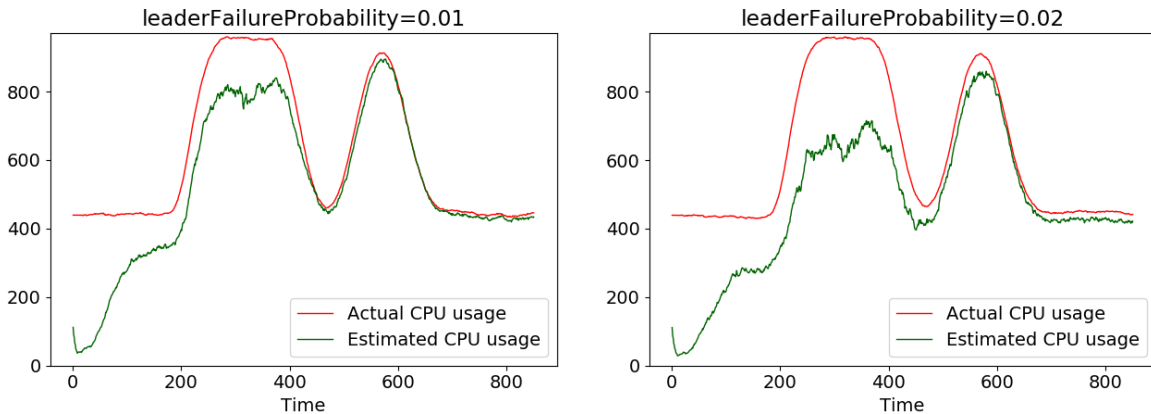
FIGURE 16. These graphs show the load estimation capability at leader nodes, for different round-wise probabilities of leader failure (leading to a disproportionate number of failures (150 to 250) in the considered timespan). The plots are obtained by taking the mean of 30 runs with different random seeds. The scenario is modelled as follows: the spikes of high load start at time $t = 200$ and $t = 500$, the new metric is injected at time $t = 450$, and we can observe that, after such upgrade, the system is able to perform a much more precise estimation of CPU usage.

settings calls for approaches (from formal languages to execution platforms) providing some abstraction of 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. Several works developed in different research communities share this attempt, often using different terminology, witnessed by various surveys, focussing on organisation of aggregates of devices [BDU$^+$13, VBD$^+$19, Cas22a], on developing frameworks for general-purpose self-organisation [MMTZ06], addressing the issue of autonomic communication [DDF$^+$06], and so on. By factoring out common ideas from these works, and neglecting the diversity in lower-level concepts, one can identify families of approaches which have relations with the programming framework developed in this paper. So, whereas the main related work is deeply covered in Section 4, in the following we comprehensively describe the research area in which our contribution can be positioned.

Device abstraction. A first family is that of languages providing some form of abstraction over device behaviour and interaction. TOTA [MZ09] and SAPERE [ZOA$^+$15] define platforms for pervasive computing focussing on agent coordination, where agents indirectly interact by injecting/perceiving "tuples" equipped with diffusion and aggregation behaviour (Java-defined in TOTA, declaratively specified by rules in SAPERE), and resulting in "fields of tuples" spread over the network; such works provide archetypal approaches to create computational fields that were an inspiration for Aggregate Computing and NC/ScaFi— though, differently from them, we provide an expressive language to better control the shape and dynamics of fields over time. Hood [WSCB04] defines data types to model an agent's neighbourhood and attributes, with operations to read/modify such attributes across neighbours, and a platform optimising execution of such operations by proper caching techniques; ScaFi provides a comparable neighbourhood abstraction, in that operator `nbr` is essentially used to declaratively access the set of neighbours as well as to combine observation of neighbours' attributes (indicated by the expression passed as argument) with modification of the same attribute locally. Finally, works such as ActorSpace [CA94], SCEL [NFLP11, ANL20], and AbC [ANL20] rely on so-called *attribute-based communication*, where each actor/agent exposes a list of attributes, and communication can be directed to the *group* of actors whose attribute match a given pattern; ScaFi can achieve a similar expressiveness with construct `branch`, by which one can define subcomputations carried on by a subset of nodes, which are those that execute the same branch and hence remain actually "observable" by operator `nbr`. Generally speaking, it is worth noting that Aggregate Computing and NC/ScaFi address the key feature of fitting 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.

Geometric/topological abstractions. Another class of related approaches falls under the umbrella of languages to express geometric constructs and topological patterns. In fact, in several application contexts concerning environment sensing and controlling, what is key is the physical (geometric, topological) shape that coalitions of mobile agents take, or that certain data items create while diffusing in the environment. In the Growing Point Language [Coo99], an *amorphous medium* [BMG08] (essentially defined by an ad-hoc network) can be programmed by a nature-inspired approach of "botanical computing", where computational processes are seen as "growing points" increasingly expanding across

neighbours until reaching a fixpoint shape defined by declarative constraints; NC/ScaFi and Aggregate Programming work on similar hypothesis on structure and behaviour of the underlying network, though adopting a different functional paradigm that is more expressive as it can address dynamical aspects as well, and which could be used as a lower-level language to reproduce the expressiveness of the growing point abstraction. The Origami Shape Language defined in [Nag08] is used to achieve similar goals of the Growing Point Language though focussing on programming a "computational surface", intended as a set of small devices working independently of their density in the surface: this language defines geometrical constructs to create basic regions and compose them, which could be turned into an API of NC/ScaFi blocks to be functionally composed to achieve similar complex geometrical structures. A more recent work is PLEIADES [BBLT18], which aim at expressing self-stabilising overlay structures in large-scale distributed systems; however, these goals are not achieved through programs as in NC/ScaFi but rather through shape formation protocols and equation-based specifications. In general, due to the universal character of field computations [ABDV18], one could consider NC/ScaFi as a viable implementation framework for a number of approaches to organise the shape of computational entities in a physical environment, with the additional byproduct of leveraging the theory of field computations to assess formal validity of certain properties, such as density independence as developed in [BVPD17] or self-stabilisation in [VAB+18].

WSN-based discovery and streaming. A number of works originating in the context of information systems for sensor networks, such as TinyDB [MSFC02], Regiment [NW04], and Cougar [YG02], address the problem of gathering information extracted from sensors in a given region of space, aggregating them somehow, and redirecting results over the devices in another region. They either focus on spatial query languages (Cougar), diffusion/aggregation policies (TinyDB) or functional models to express and manipulate streams of events (Regiment). A more recent approach is makeSense [MPO+19], with leverages graphical models of business processes compiling to an imperative object-oriented macro-programming language (mPL) which supports many-to-one report actions and many-to-many collective actions, i.e., actions possibly spanning multiple nodes. Similar approaches for mobile ad-hoc networks (MANET) have been considered that do not use data-oriented techniques but rather focus on services: SpatialViews [NKSI05] works by abstracting a MANET into *spatial views* that can be iterated on to visit nodes and request services; AmbientTalk [CBS+14] is another language for MANETs that provides resilience against transient network partitions by automatically buffering sent messages. A clear advantage of functional-based field computations as supported in NC/ScaFi is that the various bricks of information collection, aggregation, diffusion, can be defined with the same language, wrapped in homogeneous components represented by NC/ScaFi functions, and composed to create more complex applications as exemplified in Section 6, while leveraging the discovery of nearby services through an actor-based runtime.

Distributed/parallel computing. Aggregate Computing and its incarnation in ScaFi can evidently be considered as a declarative model for distributed programming, relying on abstractions and assumptions to make it easy to express certain kinds of programs by delegating important features (e.g., synthesising the concrete execution plan) to an underlying platform. As a notable example, we can draw a bridge with big data processing frameworks

like MapReduce [DG08] and its derivation Apache Spark [ZXW+16]: they essentially provide a highly declarative language of stream processing (based on simple map, reduce, filter and fold operations) and delegate to the underlying platform the duty of breaking tasks into smaller chunks to be allocated by the available computational/storage resources. An approach which is execution-wise technically more similar to the one developed in this paper is given by Bulk-Synchronous Parallel (BSP)-inspired frameworks, such as the large-scale graph processing framework Apache Giraph [SOAK16], which defines computations in terms of transformation of large graphs of data, typically stored in a distributed database. In this respect, NC/ScaFi can be seen as relying on a similar approach to achieve a more complex goal, namely, that of declaratively specifying a dynamic collective behaviour (based on distributed field computation), ultimately digesting information coming from distributed sensors and producing instructions for actuators, in such a way so as to make it breakable into smaller pieces (single rounds of computations) allocated to each device in the network.

Service choreographies. Choreographies [Pel03] are an approach to service composition where the interaction protocol of collaborative workflows is specified by a global viewpoint. So, they define the cooperative contract of multiple parties playing certain roles and collaborating to achieve a global goal. There are strong similarities with Aggregate Computing; indeed, aggregate programs *(i)* globally define the interactions supporting a collective computation, cf. the `nbr` construct; *(ii)* define services which are fundamentally collaborative in nature; and *(iii)* define roles for devices implicitly by the set of (sub-)computations executed by them or, equivalently, by the set of domain branches that they select in the collective workflow. By contrast, however, while classical choreographies usually express goal-oriented workflows where the roles are few and statically assigned to parties, aggregate computations typically carry on continuous processes where devices repeatedly participate in the collective service and can also play different roles across time depending on the context. Additionally, choreographies abstract from the particular services carried on by the involved parties, focussing instead only on when and what messages are exchanged, whereas aggregate programs specify *computations*, which are collective and yield global results represented as fields. Moreover, interaction in field computations is only possible (by alignment) between entities playing the same (sub-)computation, and is neighbour-driven (rather than peer-to-peer and role-driven). Finally, choreographies and choreographic programming [CM20] mainly focus on checking conformance or building correct-by-construction concurrent programs (e.g., deadlock-free), but do little for functional composability of adaptive behaviours, which is instead the core of Aggregate Computing.

Spatial and field-based computing languages. Aggregate Computing directly descends from 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. Notable examples include the StarLisp approach for parallel computing [LMMD88], the SDEF programming system inspired by systolic computing [EC89], and topological computing with MGS [GMCS05]. 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.

A particular class of spatial computing languages is given by *field-based coordination languages* [VBD⁺19]. Beside the field calculus, covered in this paper, other works like TOTA [MZ09] and calculi like SMuC [LLM17] also come under this umbrella. SMuC, however, does not express fields as functional compositions, but as formulas on fixpoint steps whose result may serve as continuations of the program. Specific programming languages to work with computational fields have been introduced as well, with the Proto [BB06] programming language as common ancestor, Protelis [PVB15] as its Java-oriented DSL version, and the more recent FCPP [Aud20]. The field calculus, deeply described and compared to in Section 4, has been designed as common core calculus for such languages, and for studying behavioural properties and semantic aspects [BVPD17, VAB⁺18, AVD⁺19]. Though rather similar to those languages, NC evolved in a different way: its design is profoundly influenced by the need of smoothly integrating field computations in the syntactic, semantic, and typing structures of modern, conventional languages (like Scala—see Section 5), and this required key semantic changes that motivated a more general and expressive calculus, as presented in this paper.

8. Conclusions

Aggregate Computing is a recent paradigm for "holistically" engineering CASs and smart situated ecosystems, that 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 order to promote conceptual frugality and foster smooth embedding of this programming model in mainstream languages, we propose a field calculus variant, called NC, which substitutes the notion of a "neighbouring field" with a novel notion of a "computation against a neighbour". We formalise NC and thoroughly compare to the higher-order field calculus (HFC), stressing differences in expressiveness and identifying a common fragment allowing straightforward transfer of interesting properties such as self-stabilisation. To witness the benefits of the novel calculus, we cover the ScaFi aggregate programming language, which implements NC as a DSL embedded in Scala. Finally, we use a simulated case study in edge computing to show that ScaFi/NC are effectively expressive in practice.

The availability of a Scala-based implementation of field computations naturally suggests a number of future works. From the linguistic viewpoint, it is interesting the study the interplay of field programs with advanced functional programming techniques, like the use of monads to structure the specification of increasingly complex field computations and field processes [CVA⁺19, CVA⁺21], or the use of implicit parameters to define common contexts for library functions, there included the ability of dynamically select the most proper implementation of building blocks for the application at hand [VAB⁺18]. At the ScaFi platform level, instead, we plan to deeply investigate and implement techniques for infrastructure- and QoS-aware adaptation of deployment and execution strategies for aggregate system execution, along the lines of [CPP⁺20]. This is to determine suitable application partitioning schemas and build a monitoring and control plan to effectively carry out configuration transitions. The case study we discussed in the paper also suggests the importance of collective, self-adaptive/self-organising techniques for (decentralised) edge computing and volunteer computing in ad-hoc cloudlets. In particular, a question relates to

$$
\begin{array}{cc}
[\text{E-LOC}] & [\text{E-FLD}] \quad \phi' = \phi|_{\mathbf{dom}(\Theta)\cup\{\delta\}} \\
\overline{\delta;\Theta;\sigma \vdash \ell \Downarrow \ell\langle\rangle} & \overline{\delta;\Theta;\sigma \vdash \phi \Downarrow \phi'\langle\rangle}
\end{array}
$$

$$
[\text{E-B-APP}] \quad \dfrac{
\begin{array}{c}
\delta;\pi_1(\Theta);\sigma \vdash \mathsf{e} \Downarrow \theta \quad \delta;\pi_{i+1}(\Theta);\sigma \vdash \mathsf{e}_i \Downarrow \theta_i \quad \text{for all } i \in 1,\dots,n \\
\mathsf{b} = \rho(\theta) \qquad\qquad \mathsf{v} = (\!|\mathsf{b}|\!)_\delta^{\Theta,\sigma}(\rho(\overline{\theta}))
\end{array}
}{
\delta;\Theta;\sigma \vdash \mathsf{e}(\overline{\mathsf{e}}) \Downarrow \mathsf{v}\langle\theta,\overline{\theta},\mathsf{v}\rangle
}
$$

$$
[\text{E-D-APP}] \quad \dfrac{
\begin{array}{c}
\delta;\pi_1(\Theta);\sigma \vdash \mathsf{e} \Downarrow \theta \qquad\qquad \delta;\pi_{i+1}(\Theta);\sigma \vdash \mathsf{e}_i \Downarrow \theta_i \quad \text{for all } i \in 1,\dots,n \\
\mathsf{f} = \rho(\theta) \text{ is not a built-in} \quad \delta;\pi^{\mathsf{f}}(\Theta);\sigma \vdash body(\mathsf{f})[args(\mathsf{f}) := \rho(\overline{\theta})] \Downarrow \theta'
\end{array}
}{
\delta;\Theta;\sigma \vdash \mathsf{e}(\overline{\mathsf{e}}) \Downarrow \rho(\theta')\langle\theta,\overline{\theta},\theta'\rangle
}
$$

$$
[\text{E-NBR}] \quad \dfrac{
\Theta_1 = \pi_1(\Theta) \qquad \delta;\Theta_1;\sigma \vdash \mathsf{e} \Downarrow \theta_1 \qquad \phi = \rho(\Theta_1)[\delta \mapsto \rho(\theta_1)]
}{
\delta;\Theta;\sigma \vdash \mathtt{nbr\{e\}} \Downarrow \phi\langle\theta_1\rangle
}
$$

$$
[\text{E-REP}] \quad \dfrac{
\begin{array}{ccc}
\delta;\pi_1(\Theta);\sigma \vdash \mathsf{e}_1 \Downarrow \theta_1 & \ell_1 = \rho(\theta_1) & \multirow{2}{*}{$\ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathbf{dom}(\Theta) \\ \ell_1 & \text{otherwise} \end{cases}$} \\
\delta;\pi_2(\Theta);\sigma \vdash \mathsf{e}_2[\mathsf{x} := \ell_0] \Downarrow \theta_2 & \ell_2 = \rho(\theta_2) &
\end{array}
}{
\delta;\Theta;\sigma \vdash \mathtt{rep(e_1)\{(x) \Rightarrow \{e_2\}\}} \Downarrow \ell_2\langle\theta_1,\theta_2\rangle
}
$$

FIGURE 17. Big-step rules for HFC expression evaluation.

how much such edge computing systems need orchestration, what benefits and challenges can emerge from a hybrid approach that brings self-organising processes in, and how aggregate techniques allows us to tackle such integration of paradigms.

APPENDIX A. HFC OPERATIONAL DEVICE SEMANTICS

We now present the HFC operational device semantics, modelling computation of a device within one round, as developed in literature [AVD$^+$19] and reported in Figure 17. The network semantics of HFC is not reported, since it is the same as that of NC (cf. Section 3.4). The HFC device semantics is based on value-trees and value-tree environments, as that of NC. The syntax of them, together with the auxiliary functions and syntactic shorthands, are not reported here since they are the same as in NC (cf. Figure 6). The derived judgements are slightly different, and follow the form $\delta;\Theta;\sigma \vdash \mathsf{e} \Downarrow \theta$, to be read "expression e evaluates to value-tree θ on device δ with respect to the value-tree environment Θ and sensor state σ".

Rules [E-LOC] and [E-FLD] model the evaluation of expressions that are either a local value or a neighbouring field value, respectively. For instance, evaluating the expression $\mathtt{1}$ produces (by rule [E-LOC]) the value-tree $1\langle\rangle$, while evaluating the expression $+$ produces the value-tree $+\langle\rangle$. Note that, in order to ensure that domain alignment is obeyed, rule [E-FLD] restricts the domain of the neighbouring field value ϕ to the domain of Θ augmented by δ.

Rule [E-B-APP] models the application of built-in functions. It is used to evaluate expressions of the form $\mathsf{e}(\mathsf{e}_1 \cdots \mathsf{e}_n)$ such that the evaluation of e produces a value-tree θ whose root $\rho(\theta)$ is a built-in function b. It produces the value-tree $\mathsf{v}\langle\theta,\theta_1,\dots,\theta_n,\mathsf{v}\rangle$, where $\theta_1,\dots,\theta_n$ are the value-trees produced by the evaluation of the actual parameters $\mathsf{e}_1,\dots,\mathsf{e}_n$ ($n \geq 0$) and v is the value returned by the function. Rule [E-B-APP] exploits the special auxiliary function $(\!|\mathsf{b}|\!)_\delta^{\Theta,\sigma}$, whose actual definition is abstracted away, in order to allow for

customised sets of built-in functions. This auxiliary function is such that $(\!|b|\!)_\delta^{\Theta,\sigma}(\overline{v})$ computes the result of applying built-in function b to values $\overline{v}$ in the current environment and sensor state of device δ. We require that $(\!|b|\!)_\delta^{\Theta,\sigma}(\overline{v})$ always yields values of the expected type T where b has a suitable type $(\overline{T}) \to T$.

Rule [E-D-APP] models the application of user-defined or anonymous functions, i.e., it is used to evaluate expressions of the form $e(e_1 \cdots e_n)$ such that the evaluation of e produces a value-tree θ whose root $f = \rho(\theta)$ is a user-defined function name or an anonymous function value. It is similar to rule [E-B-APP], except for the last subtree θ' of the result, which is produced by evaluating the body of the function f with respect to the value-tree environment $\pi^f(\Theta)$ containing only the value-trees associated to the evaluation of functions with the same name as f.

The evaluation of rule [E-REP] depends on whether it is performed against a tree environment with or without δ in its domain. If it is present, ℓ_0 is obtained from it (being the previously computed value for the rep construct), otherwise it is set to the result of e_1. The evaluation concludes substituting ℓ_0 for x in the body of e_2. Notice that this substitution corresponds to the result of applying $(x) \Rightarrow \{e_2\}(\ell_0)$ according to rule [E-D-APP] (skipping some branches of the resulting value-tree).

Value-trees also support modelling information exchange through the nbr construct, as of rule [E-NBR]. In this rule, the neighbours' values for e are extracted into a neighbouring field value as $\phi = \rho(\Theta_1)$. Then $\phi(\delta)$ is updated to the more recent value $\ell = \rho(\theta_1)$, as represented by the notation $\phi[\delta \mapsto \ell]$.

Appendix B. Proofs

B.1. Computation Determinism.

Restatement of Lemma 3.15 (Computation Determinism). *Let e be a well-typed closed expression, $\Theta \in WFVTE(\mathcal{D}; \mathcal{A}; e)$. Then for all device identifiers δ, δ' and sensor state σ:*

(1) *$\delta, \delta; \Theta; \sigma \vdash e$ FAIL cannot hold.*
(2) *There is at most one derivation of the kind $\delta, \delta'; \Theta; \sigma \vdash e \Downarrow \theta$ or $\delta, \delta'; \Theta; \sigma \vdash e$ FAIL.*

Proof. (1) Notice that a failure can occur only if in a certain subexpression (which is a relational sensor application or nbr-expression) the neighbour δ' is not in $\mathbf{dom}(\Theta) \cup \{\delta\}$, thus in particular $\delta' \neq \delta$. Any construct other than a $foldhood$ either propagates the neighbour δ' as is or resets it to δ, so for a failure to occur in a sub-evaluation of $\delta, \delta; \Theta; \sigma \vdash e$ FAIL the involved subexpression must be inside a $foldhood$-expression. However, $foldhood$-expressions never fail, as they "absorb" failures by skipping failing neighbours, concluding the proof.

(2) We assume that e has at least one possible derivation of the big-step operational semantics, and prove by induction on its length that this derivation is unique.

If e is a value, a rep- or $foldhood$-expression, then it cannot fail since the conclusions of every rule in Figure 7 is either a function application or a nbr-expression. Furthermore, its evaluation produces a value by rules [E-VAL], [E-REP] or [E-FOLD] (respectively), using fact (1) and observing that the only sub-evaluation which may fail occur in the third argument of a $foldhood$, and does not influence the evaluation of the $foldhood$ itself.

Such value must be unique, since by inductive hypothesis so are the results of its subexpressions and rules are deterministic.

If e is an nbr-expression, then exactly one of the rules [E-NBR], [E-NBR-LOC] or [E-NBR-FAIL] must be applicable, depending on whether resp. $\delta' = \delta$, $\delta' \in \mathbf{dom}(\Theta) \setminus \delta$, or neither holds. In the first two cases, the result is then unique by inductive hypothesis and rule determinism.

Finally, if $e = e'(\bar{e})$ is a function application, then the evaluation of e' does not fail by fact (1), and thus produces a uniquely determined function f (with a possibly infinite derivation) by inductive hypothesis. If the evaluation of some of the arguments fails, then rule [E-APP-ARG-FAIL] applies and e fails, and none of the other function application rules ([E-B-APP], [E-D-APP], [E-R-APP-FAIL], [E-D-APP-FAIL]) applies since they all require all arguments to evaluate to a value.

Otherwise, $\bar{e}$ evaluates (uniquely) to $\bar{v}$. If $f = r$ is a relational sensor and δ' is not in $\mathbf{dom}(\pi^{r}(\Theta)) \cup \{\delta\}$, then e fails by Rule [E-R-APP-FAIL] and does not produce a value since $(\!(r)\!)_{\delta,\delta'}^{\pi^{r}(\Theta),\sigma}$ is undefined making rule [E-B-APP] inapplicable. If instead f is a built-in function which is not a relational sensor or δ' is in $\mathbf{dom}(\pi^{f}(\Theta)) \cup \{\delta\}$, then $(\!(f)\!)_{\delta,\delta'}^{\pi^{f}(\Theta),\sigma}$ is defined and rule [E-B-APP] is applicable while rule [E-R-APP-FAIL] is not. The result is then unique by inductive hypothesis and rule determinism.

The only remaining case is when f is a user-defined or anonymous function. By inductive hypothesis, $body(f)[args(f) := \rho(\bar{\theta})]$ either fails or it produces a unique value. In the first case, rule [E-D-APP] applies (producing an unique value) while rule [E-D-APP-FAIL] does not; in the second case the converse happens, concluding that derivations are unique (when present). □

B.2. Calculus Equivalence.

Restatement of Theorem 4.7 (Equivalence between HFC$'$ and NC$'$). *Assume that for every built-in function* b *with local arguments,* $b(\bar{x})$ *has the same behaviour in HFC$'$ and NC$'$ for every substitution of* $\bar{x}$. *Assume that* $\mathcal{D}; \mathcal{A} \vdash e : T$ *in HFC$'$/NC$'$ where* $\mathcal{A} = \bar{x} : \overline{T}$, *let* $\bar{v}$ *be HFC$'$ values of type* $\overline{T}$. *Then* $e[\bar{x} := \bar{v}]$ *has the same behaviour in HFC$'$ and NC$'$.*

Proof. We proceed by induction on the syntax of closed expressions e, simultaneously for all possible network environments.
- e = v: since NC rule [E-VAL] is identical to HFC rule [E-LOC], the thesis holds whenever e = v is a local value. Since field values are not allowed to appear in source HFC programs, and do not exist in NC programs, the thesis follows.
- e = x: if T is a local type, the same reasoning for e = v applies. If T is a field type and x is substituted with ϕ, the evaluation result is $\phi|_{\mathbf{dom}(\Theta)}$ in HFC. In NC, for every $\delta \in \mathbf{dom}(\Theta)$, $e[\bar{x} := \bar{v}(\delta)]$ is $\phi(\delta)$, which is a local value hence it evaluates to itself by rule [E-VAL], concluding this part of the proof.
- $e = e_{n+1}(\bar{e})$: notice that rules [E-B-APP] and [E-D-APP] are identical in HFC and NC (ignoring the neighbour device δ'). This concludes the proof in case all $\bar{e}$ and the resulting output of e_{n+1} are local. Otherwise, there are three possibilities:
  - e_{n+1} evaluates to a built-in function, so that the thesis follows by inductive hypothesis and the coherence hypothesis on built-in functions.

– e_{n+1} evaluates to a user-defined or anonymous function d. In this case, the domain of computation is restricted both in HFC and in NC to the aligned neighbours $\pi^d(\Theta)$; which in fact reduces all fields involved in the computation to the new domain, both in HFC by Rule [E-FLD] and in NC by Rule [E-FOLD] (which aggregates only values from devices in $\mathbf{dom}(\Theta)$).

After alignment, in both cases the argument's values are substituted into the body of d. If all arguments have local type, the substitution is performed in the same way both in HFC and NC, hence the result of e corresponds by inductive hypothesis. If some argument has field type, then those arguments cannot occur within folding expressions, and are either ignored or manipulated point-wise to form a field result. In this case, the results of e for every neighbour δ' correspond to the point-wise results of field in HFC, concluding the proof in this case.

- $e = \texttt{rep}(e_1)\{(x)\texttt{=>}\{e_2\}\}$: the thesis follows (with a further induction on firing events) by noticing that rule [E-REP] in HFC corresponds to that of NC together with the expansion of the anonymous function application.
- $e = \texttt{nbr}\{e_1\}$: by inductive hypothesis, e_1 is an expression of local type which always evaluates to the same trees in HFC and NC. It follows by HFC rule [E-NBR] that e in δ evaluates to the field mapping $\overline{\delta}$ to the corresponding values $\overline{v}$ of e_1 in those devices. By NC rules [E-NBR] (for $\delta_i \neq \delta$) and [E-NBR-LOC] (for $\delta_i = \delta$) it follows that $\delta, \delta_i; \Theta; \sigma \vdash e \Downarrow v_i \langle \overline{\theta} \rangle$ for all $i = 1 \dots n$.
- $e = \texttt{foldhood}(e_1, e_2, e_3)$: by inductive hypothesis, e_1 and e_2 evaluate to the same local values v', f while e_3 does not contain free variables of field type and it satisfies that

$$\delta; \Theta; \sigma \vdash e_3 \Downarrow (\overline{\delta} \mapsto \overline{v})\langle \overline{\theta} \rangle \iff \delta, \overline{\delta}; \Theta; \sigma \vdash e_3 \Downarrow \overline{v}\langle \overline{\theta} \rangle$$

By NC rule [E-FOLD], the overall result of the expression is then $f(v', \overline{v})$ (computed two elements at a time), which corresponds to the result of applying $\texttt{foldhood}$ in HFC. $\qquad\square$

Acknowledgment

References

[ABD+20] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. Field-based coordination with the share operator. *Log. Methods Comput. Sci.*, 16(4), 2020. `doi: 10.23638/LMCS-16(4:1)2020`.

[ABDV18] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. Space-time universality of field calculus. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages*, volume 10852 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2018. `doi:10.1007/978-3-319-92408-3_1`.

[ABE+13] Stuart Anderson, Nicolas Bredeche, A.E. Eiben, George Kampis, and Maarten van Steen. *Adaptive collective systems: herding black sheep*. Bookprints, 2013.

[ACD+21] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. Optimal resilient distributed data collection in mobile edge environments. *Comput. Electr. Eng.*, 96(Part):107580, 2021. `doi:10.1016/j.compeleceng.2021.107580`.

[ACDV17]   Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. Compositional blocks for optimal self-healing gradients. In *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, September 18-22, 2017*, pages 91–100. IEEE Computer Society, 2017. `doi:10.1109/SASO.2017.18`.

[ADVB18]   Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Enrico Bini. Distributed real-time shortest-paths computations with the field calculus. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 23–34, 2018. `doi:10.1109/RTSS.2018.00013`.

[ADVC16]   Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Roberto Casadei. Run-time management of computation domains in field calculus. In *Foundations and Applications of Self* Systems, IEEE International Workshops on*, pages 192–197. IEEE, 2016. `doi:10.1109/FAS-W.2016.50`.

[AHKY15]   Cyrille Artho, Klaus Havelund, Rahul Kumar, and Yoriyuki Yamagata. Domain-specific languages with Scala. In *ICFEM*, volume 9407 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2015. `doi:10.1007/978-3-319-25423-4_1`.

[ANL20]   Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. Programming interactions in collective adaptive systems by relying on attribute-based communication. *Sci. Comput. Program.*, 192:102428, 2020. `doi:10.1016/j.scico.2020.102428`.

[Aud20]   Giorgio Audrito. FCPP: an efficient and extensible field calculus framework. In *Proceedings of the 1st International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS*, pages 153–159. IEEE Computer Society, 2020. `doi:10.1109/ACSOS49614.2020.00037`.

[AVD+19]   Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *ACM Trans. Comput. Log.*, 20(1):5:1–5:55, 2019. `doi:10.1145/3285956`.

[BB06]   Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intell. Syst.*, 21(2):10–19, 2006. `doi:10.1109/MIS.2006.29`.

[BBLT18]   Simon Bouget, Yérom-David Bromberg, Adrien Luxey, and François Taïani. Pleiades: Distributed structural invariants at scale. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 542–553. IEEE Computer Society, 2018. `doi:10.1109/DSN.2018.00062`.

[BDT99]   Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence - From Natural to Artificial Systems*. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press, 1999. URL: `http://ukcatalogue.oup.com/product/9780195131598.do`.

[BDU+13]   Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. In Marjan Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013. A longer version available at: `http://arxiv.org/abs/1202.5509`. `doi:10.4018/978-1-4666-2092-6.ch016`.

[Bet16]   Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Birmingham, ISBN: 9781786464965, Packt Publishing Ltd., UK, 2016.

[BMG08]   Jonathan Bachrach, James McLurkin, and Anthony Grue. Protoswarm: a language for programming multi-robot systems using the amorphous medium abstraction. In Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons, editors, *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Volume 3*, pages 1175–1178. IFAAMAS, 2008. URL: `https://dl.acm.org/citation.cfm?id=1402824`.

[BPV15]   Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015. `doi:10.1109/MC.2015.261`.

[BV14]   Jacob Beal and Mirko Viroli. Building blocks for aggregate programming of self-organising applications. In *Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, United Kingdom, September 8-12, 2014*, pages 8–13. IEEE Computer Society, 2014. `doi:10.1109/SASOW.2014.6`.

[BVPD17]   Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution in the Internet of Things. *ACM Trans. Auton. Adapt. Syst.*, 12(3):12:1–12:29, September 2017. `doi:10.1145/3105758`.

[CA94]   Christian J Callsen and Gul Agha. Open heterogeneous computing in actorspace. *J. Parallel Distributed Comput.*, 21(3):289–300, 1994. `doi:10.1006/jpdc.1994.1060`.

[Cas22a]   Roberto Casadei. Macroprogramming: Concepts, state of the art, and opportunities of macro-scopic behaviour modelling. *CoRR*, abs/2201.03473, 2022. URL: `https://arxiv.org/abs/2201.03473`, `arXiv:2201.03473`.

[Cas22b]   Roberto Casadei. metaphori/experiments-neighbours-calculus: v1.0, 2022. `doi:10.5281/ZENODO.7220054`.

[CAV18]   Roberto Casadei, Alessandro Aldini, and Mirko Viroli. Towards attack-resistant aggregate computing using trust mechanisms. *Sci. Comput. Program.*, 2018. `doi:10.1016/j.scico.2018.07.006`.

[CBS⁺14]   Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Comput. Lang. Syst. Struct.*, 40(3-4):112–136, 2014. `doi:10.1016/j.cl.2014.05.002`.

[CDM⁺10]   Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 835–847. ACM, 2010. `doi:10.1145/1869459.1869527`.

[CFP⁺22]   Roberto Casadei, Giancarlo Fortino, Danilo Pianini, Andrea Placuzzi, Claudio Savaglio, and Mirko Viroli. A methodology and simulation-based toolchain for estimating deployment performance of smart collective services at the edge. *IEEE Internet Things J.*, 9(20):20136–20148, 2022. `doi:10.1109/JIOT.2022.3172470`.

[CHL⁺15]   Min Chen, Yixue Hao, Yong Li, Chin-Feng Lai, and Di Wu. On the computation offloading at ad hoc cloudlet: architecture and service modes. *IEEE Commun. Mag.*, 53(6-Supplement):18–24, 2015. `doi:10.1109/MCOM.2015.7120041`.

[Chu32]   Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932. `doi:10.2307/1968337`.

[CM20]   Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. `doi:10.1016/j.tcs.2019.07.005`.

[Coo99]   Daniel Coore. *Botanical computing: a developmental approach to generating interconnect topologies on an amorphous computer*. PhD thesis, Massachusetts Institute of Technology, 1999.

[CPP⁺20]   Roberto Casadei, Danilo Pianini, Andrea Placuzzi, Mirko Viroli, and Danny Weyns. Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment. *Future Internet*, 12(11):203, 2020. `doi:10.3390/fi12110203`.

[CPSV19]   Roberto Casadei, Danilo Pianini, Guido Salvaneschi, and Mirko Viroli. On context-orientation in aggregate programming. In *IEEE 4th International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2019, Umea, Sweden, June 16-20, 2019*, pages 92–97. IEEE, 2019. `doi:10.1109/FAS-W.2019.00035`.

[CPVN19]   Roberto Casadei, Danilo Pianini, Mirko Viroli, and Antonio Natali. Self-organising coordination regions: A pattern for edge computing. In Hanne Riis Nielson and Emilio Tuosto, editors, *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11533 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2019. `doi:10.1007/978-3-030-22397-7_11`.

[CRD⁺17]   Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, and Frank Piessens. FRP IoT modules as a Scala DSL. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2017, pages 15–20, New York, NY, USA, 2017. ACM. `doi:10.1145/3141858.3141861`.

[CTVD19]   Roberto Casadei, Christos Tsigkanos, Mirko Viroli, and Schahram Dustdar. Engineering resilient collaborative edge-enabled IoT. In *2019 IEEE International Conference on Services Computing (SCC)*, pages 36–45, July 2019. `doi:10.1109/SCC.2019.00019`.

[CV18]   Roberto Casadei and Mirko Viroli. Programming actor-based collective adaptive systems. In Alessandro Ricci and Philipp Haller, editors, *Programming with Actors: State-of-the-Art and*

Research Perspectives, volume 10789 of *Lecture Notes in Computer Science*, pages 94–122. Springer International Publishing, 2018. `doi:10.1007/978-3-030-00302-9_4`.

[CV19]   Roberto Casadei and Mirko Viroli. Coordinating computation at the edge: a decentralized, self-organizing, spatial approach. In *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 60–67, June 2019. `doi:10.1109/FMEC.2019.8795355`.

[CVA⁺19]   Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. Aggregate processes in field calculus. In Hanne Riis Nielson and Emilio Tuosto, editors, *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11533 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2019. `doi:10.1007/978-3-030-22397-7_12`.

[CVA⁺21]   Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. Engineering collective intelligence at the edge with aggregate processes. *Engineering Applications of Artificial Intelligence*, 97:104081, 2021. `doi:10.1016/j.engappai.2020.104081`.

[DDF⁺06]   Simon Dobson, Spyros G. Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, 2006. `doi:10.1145/1186778.1186782`.

[DG08]   Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. `doi:10.1145/1327452.1327492`.

[DM82]   Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages*, POPL '82, pages 207–212. ACM, 1982. `doi:10.1145/582153.582176`.

[Doe18]   Sébastien Doeraene. *Cross-Platform Language Design*. PhD thesis, EPFL, Switzerland, 2018. `doi:10.5075/epfl-thesis-8733`.

[DS14]   Muhammad Nouman Durrani and Jawwad A. Shamsi. Volunteer computing: requirements, challenges, and solutions. *J. Netw. Comput. Appl.*, 39:369–380, 2014. `doi:10.1016/j.jnca.2013.07.006`.

[Dub11]   Gilles Dubochet. *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. PhD thesis, EPFL, Switzerland, 2011. `doi:10.5075/epfl-thesis-5007`.

[EC89]   Bradley R. Engstrom and Peter R. Cappello. The SDEF programming system. *J. Parallel Distributed Comput.*, 7(2):201–231, 1989. `doi:10.1016/0743-7315(89)90018-X`.

[Gho11]   Debasish Ghosh. DSL for the uninitiated. *Commun. ACM*, 54(7):44–50, 2011. `doi:10.1145/1965724.1965740`.

[GMCS05]   Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computations in space and space in computations. In *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 137–152. Springer, Berlin, 2005.

[IPW01]   Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. `doi:10.1145/503502.503505`.

[LLM17]   Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. Asynchronous distributed execution of fixpoint-based computational fields. *Log. Methods Comput. Sci.*, 13(1), 2017. `doi:10.23638/LMCS-13(1:13)2017`.

[LMMD88]   C. Lasser, J.P. Massar, J. Miney, and L. Dayton. *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.

[MMTZ06]   Marco Mamei, Ronaldo Menezes, Robert Tolksdorf, and Franco Zambonelli. Case studies for self-organization in computer science. *J. Syst. Archit.*, 52(8-9):443–460, 2006. `doi:10.1016/j.sysarc.2006.02.002`.

[MPO⁺19]   Luca Mottola, Gian Pietro Picco, Felix Jonathan Oppermann, Joakim Eriksson, Niclas Finne, Harald Fuchs, Andrea Gaglione, Stamatis Karnouskos, Patricio Moreno Montero, Nina Oertel, Kay Römer, Patrik Spieß, Stefano Tranquillini, and Thiemo Voigt. makesense: Simplifying the integration of wireless sensor networks into business processes. *IEEE Trans. Software Eng.*, 45(6):576–596, 2019. `doi:10.1109/TSE.2017.2787585`.

[MSFC02] Samuel Madden, Robert Szewczyk, Michael J. Franklin, and David E. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002), 20-21 June 2002, Callicoon, NY, USA*, pages 49–58. IEEE Computer Society, 2002. `doi:10.1109/MCSA.2002.1017485`.

[MZ09]   Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4):1–56, 2009. `doi:http://doi.acm.org/10.1145/1538942.1538945`.

[Nag08]   Radhika Nagpal. Programmable pattern-formation and scale-independence. In *Unifying Themes in Complex Systems IV*, pages 275–282. Springer Berlin Heidelberg, 2008. `doi:10.1007/978-3-540-73849-7_31`.

[NFLP11] Rocco De Nicola, Gian Luigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects, 10th International Symposium, FMCO 2011, Turin, Italy, October 3-5, 2011, Revised Selected Papers*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48. Springer, 2011. `doi:10.1007/978-3-642-35887-6_2`.

[NJW20]   Rocco De Nicola, Stefan Jähnichen, and Martin Wirsing. Rigorous engineering of collective adaptive systems: special section. *Int. J. Softw. Tools Technol. Transf.*, 22(4):389–397, 2020. `doi:10.1007/s10009-020-00565-0`.

[NKSI05] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. *ACM SIGPLAN Notices*, 40(6):249–260, 2005. `doi:10.1145/1064978.1065040`.

[NW04]   Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In Alexandros Labrinidis and Samuel Madden, editors, *Proceedings of the 1st Workshop on Data Management for Sensor Networks, in conjunction with VLDB, DMSN 2004, Toronto, Canada, August 30, 2004*, volume 72 of *ACM International Conference Proceeding Series*, pages 78–87. ACM, 2004. `doi:10.1145/1052199.1052213`.

[OMO10]   Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *SIGPLAN Not.*, 45(10):341–360, oct 2010. `doi:10.1145/1932682.1869489`.

[OR14]   Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Commun. ACM*, 57(4):76–86, 2014. `doi:10.1145/2591013`.

[OZ05]   Martin Odersky and Matthias Zenger. Scalable component abstractions. *ACM SIGPLAN Notices*, 40(10):41, October 2005. `doi:10.1145/1103845.1094815`.

[PCV+21] Danilo Pianini, Roberto Casadei, Mirko Viroli, Stefano Mariani, and Franco Zambonelli. Time-fluid field-based coordination through programmable distributed schedulers. *Log. Methods Comput. Sci.*, 17(4), 2021. `doi:10.46298/lmcs-17(4:13)2021`.

[Pel03]   Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003. `doi:10.1109/MC.2003.1236471`.

[PMV13]   Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with Alchemist. *J. Simulation*, 2013. `doi:10.1057/jos.2012.27`.

[PVB15]   Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1846–1853. ACM, 2015. `doi:10.1145/2695664.2695913`.

[Rit18]   Pierluigi Riti. *External DSL*, pages 59–69. Apress, Berkeley, CA, 2018. `doi:10.1007/978-1-4842-3036-7_4`.

[SCZ+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet Things J.*, 3(5):637–646, 2016. `doi:10.1109/JIOT.2016.2579198`.

[SGP12]   Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *J. Syst. Softw.*, 85(8):1801–1817, 2012. `doi:10.1016/j.jss.2012.03.024`.

[Slo08]   Tony Sloane. Experiences with Domain-specific Language Embedding in Scala. In Julia Lawall and Laurent Réveillère, editors, *Domain-Specific Program Development*, page 7, Nashville, United States, 2008. URL: `https://hal.archives-ouvertes.fr/hal-00350269`.

[SOAK16]   Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. *Large-Scale Graph Processing Using Apache Giraph.* Springer, 2016. `doi:10.1007/978-3-319-47431-1`.

[Szu01]   Tadeusz M Szuba. *Computational collective intelligence.* John Wiley & Sons, Inc., 2001. URL: `https://books.google.it/books?id=YINQAAAAMAAJ`.

[VAA+18]   Karima Velasquez, David Perez Abreu, Marcio R. M. Assis, Carlos R. Senna, Diego F. Aranha, Luiz F. Bittencourt, Nuno Laranjeiro, Marília Curado, Marco Vieira, Edmundo Monteiro, and Edmundo Roberto Mauro Madeira. Fog orchestration for the Internet of Everything: state-of-the-art and research challenges. *J. Internet Serv. Appl.*, 9(1):14:1–14:23, 2018. `doi:10.1186/s13174-018-0086-3`.

[VAB+18]   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):16:1–16:28, 2018. `doi:10.1145/3177774`.

[VBD+19]   Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *J. Log. Algebraic Methods Program.*, 109, 2019. `doi:10.1016/j.jlamp.2019.100486`.

[VBDP15]   Mirko Viroli, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. 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*, pages 81–90. IEEE, Sept 2015. `doi:10.1109/SASO.2015.16`.

[VCP16a]   Mirko Viroli, Roberto Casadei, and Danilo Pianini. On execution platforms for large-scale aggregate computing. In Paul Lukowicz, Antonio Krüger, Andreas Bulling, Youn-Kyung Lim, and Shwetak N. Patel, editors, *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp Adjunct 2016, Heidelberg, Germany, September 12-16, 2016*, pages 1321–1326. ACM, 2016. `doi:10.1145/2968219.2979129`.

[VCP16b]   Mirko Viroli, Roberto Casadei, and Danilo Pianini. Simulating large-scale aggregate MASs with Alchemist and Scala. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016*, volume 8 of *Annals of Computer Science and Information Systems*, pages 1495–1504. IEEE, 2016. `doi:10.15439/2016F407`.

[Voe13]   M. Voelter. *DSL Engineering: Designing, Implementing and Using Domain-specific Languages.* CreateSpace Independent Publishing Platform, 2013. URL: `https://books.google.it/books?id=J2i0lwEACAAJ`.

[WSCB04]   Kamin Whitehouse, Cory Sharp, David E. Culler, and Eric A. Brewer. Hood: A neighborhood abstraction for sensor networks. In Guruduth S. Banavar, Willy Zwaenepoel, Doug Terry, and Roy Want, editors, *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services, MobiSys 2004, Hyatt Harborside, Boston, Massachusetts, USA, June 6-9, 2004.* ACM / USENIX, 2004. `doi:10.1145/990064.990079`.

[YG02]   Yong Yao and Johannes Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002. `doi:10.1145/601858.601861`.

[ZOA+15]   Franco Zambonelli, Andrea Omicini, Bernhard Anzengruber, Gabriella Castelli, Francesco L. De Angelis, Giovanna Di Marzo Serugendo, Simon Dobson, Jose Luis Fernandez-Marquez, Alois Ferscha, Marco Mamei, Stefano Mariani, Ambra Molesini, Sara Montagna, Jussi Nieminen, Danilo Pianini, Matteo Risoldi, Alberto Rosi, Graeme Stevenson, Mirko Viroli, and Juan Ye. Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive and Mobile Computing*, 17, Part B(0):236 – 252, 2015. 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian. `doi:10.1016/j.pmcj.2014.12.002`.

[ZXW+16]   Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016. `doi:10.1145/2934664`.