ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

A Higher-Order Calculus of Computational Fields

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J. (2019). A Higher-Order Calculus of Computational Fields. ACM TRANSACTIONS ON COMPUTATIONAL LOGIC, 20(1), 1-55 [10.1145/3285956].

Availability:
This version is available at: https://hdl.handle.net/11585/686120 since: 2019-05-06

Published:
DOI: http://doi.org/10.1145/3285956

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2024

This is the final peer-reviewed accepted manuscript of:

Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. 2019. A Higher-Order Calculus of Computational Fields. *ACM Trans. Comput. Logic* 20, 1, Article 5 (February 2019), 55 pages. DOI:https://doi.org/10.1145/3285956

The final published version is available online at: **https://doi.org/10.1145/3285956**

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

A Higher-order Calculus of Computational Fields*

Mirko Viroli, Giorgio Audrito, Ferruccio Damiani, Danilo Pianini, Jacob Beal

Thursday 23rd August, 2018

Abstract

The complexity of large-scale distributed systems, particularly when deployed in physical space, calls for new mechanisms to address composability and reusability of collective adaptive behaviour. Computational fields have been proposed as an effective abstraction to fill the gap between the macro-level of such systems (specifying a system's collective behaviour) and the micro-level (individual devices' actions of computation and interaction to implement that collective specification), thereby providing a basis to better facilitate the engineering of collective APIs and complex systems at higher levels of abstraction. This paper proposes a full formal foundation for field computations, in terms of a core (higher-order) calculus of computational fields containing a few key syntactic constructs, and equipped with typing, denotational and operational semantics. Critically, this allows formal establishment of a link between the micro- and macro-levels of collective adaptive systems, by a result of full abstraction and adequacy for the (aggregate) denotational semantics with respect to the (per-device) operational semantics.

1 Introduction

The increasing availability of computational devices of every sort, spread throughout our living and working environments, is transforming the challenges in construction of complex software applications, particularly if we wish them to take full opportunity of this computational infrastructure. Large scale, heterogeneity of communication infrastructure, need for resilience to unpredictable changes, openness to on-the-fly adoption of new code and behaviour, and pervasive collectiveness in sensing, planning and actuation: all these features will soon be the norm in a great variety of scenarios of pervasive computing, the Internet-of-Things, cyber-physical systems, etc. Currently, however, it is extremely difficult to engineer collective applications of this kind, mainly due to the lack of computational frameworks well suited to deal with this level of complexity in application services. Most specifically, there is need to provide mechanisms by which reusability and composability of components for collective adaptive behaviour becomes natural and implicit, such that they can support the construction of layered APIs with formal behaviour guarantees, sufficient to readily enable the creation of complex applications.

Aggregate computing [5] is a paradigm aiming to address this problem, by means of the notion of *computational field* [31] (or simply *field*): this is a global, distributed map from computational devices to computational objects (data values of any sort, including higher-order objects such as functions and processes). Computing with fields means computing such global structures, and defining a reusable block of behaviour means to define a reusable computation from fields to fields: this functional view holds at any level of abstraction, from low-level mechanisms up to whole applications, which ultimately work by getting input fields from sensors and process them to produce output fields to actuators.

The *field calculus* [18, 17] is a tiny functional language providing basic constructs to work with fields, whose operational semantics can act as blueprint for actual implementations where myriad devices interact via proximity-based broadcasts. Field calculus provides a unifying approach to understanding and analysing the wide range of approaches to distributed systems engineering that make use of computational fields [4]. Recent works have also adopted this field calculus as a *lingua franca* to investigate formal properties of resiliency to environment changes [37, 43], and to device distribution [7].

*Author's addresses: M. Viroli and D. Pianini, DISI, University of Bologna, Italy; G.Audrito and F. Damiani, Dipartimento di Informatica, University of Torino, Italy; J. Beal, Raytheon BBN Technologies, USA.

In this paper we propose a full foundation for aggregate computing and field computations. We introduce syntax, typing, denotational semantics, properties, and operational semantics of a higher-order version of field calculus, where functions—and hence, computational behaviour—can be seen as objects amenable to manipulation just like any other data structure, and can hence be injected at run-time during system operation through sensors, spread around, and be executed by all (or some) devices, which then coordinate on the collective computation of a new service.

A key insight and technical result of this paper is that the notoriously difficult problem of reconciling local and global behaviour in a complex adaptive system [4] can be connected to a well-known problem in programming languages: correspondence between denotational and operational semantics. On the one hand, in field calculus, denotational semantics characterises computations in terms of their global effect across space (available devices) and time (device computation events)—i.e., the macro level. On the other hand, operational semantics gives a transition system dictating each device's individual and local computing/interactive behaviour—i.e., the micro level. Correspondence between the two, formally proved in this paper via *adequacy* and *full abstraction* (c.f., [15, 42]), thus provides a formal micro-macro connection: one designs a system considering the denotational semantics of programming constructs, and an underlying platform running the distributed interpreter defined by the operational semantics guarantees a consistent execution. This is a significant step towards effective methods for the engineering of self-adaptive systems, achieved thanks to the standard theory and framework of programming languages.

The remainder of this paper is organised as follows: Section 2 reviews related works and the background for this work, describing the key elements of higher-order field computations. Section 3 defines syntax and typing of the proposed calculus, which is then provided with two semantics: Section 4 defines denotational semantics, while Section 5 defines operational semantics. Section 6 then discusses and proves properties of these semantics, including adequacy and full abstraction, and Section 7 gives examples showing the expressive power of the proposed calculus for engineering collective adaptive systems. Finally Section 8 concludes and discusses future directions.[1]

2 Related Work and Background

The work on field calculus presented in this paper builds on a sizable body of prior work. In this section, we begin with a general review of work on the programming of aggregates. Following this, we aim to provide the reader with examples and intuition that can aid in understanding the more formal presentation in subsequent sections, presenting a conceptual introduction to programming with fields and the extension of these concepts to first-class functions over fields.

2.1 Macro-programming and the aggregation problem

One of the key challenges in software engineering for collective adaptive systems is that such systems frequently comprise a potentially high number of devices (or agents) that need to interact locally (e.g., interacting by proximity as in wireless sensor networks), either of necessity or for the sake of efficiency. Such systems need to carry on their collective tasks cooperatively, and to leverage such cooperation in order to adapt to unexpected contingencies such as device failures, loss of messages, changes to inputs, modification of network topology, etc. Engineering locally-communicating collective systems has long been a subject of interest in a wide variety of fields, from biology to robotics, from networking to high-performance computing, and many more.

Despite the diversity of fields involved, however, a uniting has been the search for appropriate mechanisms, models, languages and tools to organise cooperative computations as carried out by a potentially vast aggregation of devices spread over space.

A general survey of work in this area may be found in [4], which we summarise and complement here. Across the multitude of approaches that have been developed in the past, a number of common themes have emerged, and prior approaches may generally be understood as falling into one of several clusters in alignment with these themes:

[1] This paper is an extended version of the work in [18], adding: a reduced (yet more expressive) and reworked set of constructs, a type system, denotational semantics, and adequacy and full abstraction results.

- *Foundational approaches to group interaction:* These approaches present mathematically concise foundations for capturing the interaction of groups in complex environments, most often by extending the archetypal process algebra π-calculus, which originally models flat compositions of processes. Such approaches include various models of environment structure (from "ambients" to 3D abstractions) [10, 11, 34], shared-space abstractions by which multiple processes can interact in a decoupled way [8, 44], and attribute-based models declaratively specifying the target of communication so as to dynamically create ensembles [20].

- *Device abstraction languages:* These approaches allow a programmer to focus on cooperation and adaptation by making the details of device interactions implicit. For instance, TOTA [31] allows one to program tuples with reaction and diffusion rules, while in the SAPERE approach [48] such rules are embedded in space and apply semantically, and the $\sigma\tau$-Linda model [47] manipulates tuples over space and time. Other examples include MPI [33], which declaratively expresses topologies of processes in supercomputing applications, NetLogo [41], which provides abstract means to interact with neighbours following the cellular automata style, and Hood [49], which implicitly shares values with neighbours;

- *Pattern languages:* These approaches provide adaptive means for composing geometric and/or topological constructions, though with little focus on computational capability. For example, the Origami Shape Language [35] allows the programmer to imperatively specify geometric folds that are compiled into processes identifying regions of space, Growing Point Language [14] provides means to describe topologies in terms of a "botanical" metaphor with growing points and tropisms, ASCAPE [27] supports agent communication by means of topological abstractions and a rule language, and the catalogue of self-organisation patterns in [21] organises a variety of mechanisms from low-level primitives to complex self-organization patterns.

- *Information movement languages:* These are the complement of pattern languages, providing means for summarising information obtained from across space-time regions of the environment and streaming these summaries to other regions, but little control over the patterning of that computation. Examples include TinyDB [30] viewing a wireless sensor network as a database, Regiment [36] using a functional language to be compiled into protocols of device-to-device interaction, and the agent communication language KQML [22].

- *Spatial computing languages:* These provide flexible mechanisms and abstractions to explicitly consider spatial aspects of computation, avoiding the limiting constraints of the other categories. For example, Proto [3] is a Lisp-like functional language and simulator for programming wireless sensor networks with the notion of computational fields, and MGS [25] is a rule-based language for computation of and on top of topological complexes.

Overall, the successes and failures of these language suggest, as observed in [5], that adaptive mechanisms are best arranged to be implicit by default, that composition of aggregate-level modules and subsystems must be simple, transparent, and result in highly predictable behaviours, and that large-scale collective adaptive systems typically require a mixture of coordination mechanisms to be deployed at different places, times, and scales.

2.2 Computing with fields

At the core of the approach we present in this paper is a shift from individual devices computing single values, to whole networks computing *fields*, where a field is a collective structure that maps each device in some portion of the network to locally computed values over time. Accordingly, instead of considering computation as a process of manipulating input events to produce output events, computing with fields means to take fields as inputs and produce fields as outputs.

This change of focus has a deep impact when it comes to the engineering of complex applications for large networks of devices, in which it is important that the identity and position of individual devices should not exert a significant influence on the operation of the system as a whole. Applying the field approach to building such systems, one can create reusable distributed algorithms and define functions (from fields to

fields) as building blocks, structure such building blocks into libraries of increasing complexity, and compose them to create whole application services [5].

For example, assume that one is able to define the following three functions:

- `distance-to(source)`: This function takes an indicator field `source` of Boolean values, holding true at a set of devices considered as *sources*, and yields a field of real values, estimating shortest distance from each device to the closest source (if each device is assumed capable of locally estimating distance to close neighbours, long-range distance estimates can be computed transitively).

- `converge-sum(potential,val)`: This function takes a field `potential` of real values, and a field `val` of numeric values, and it accumulates all values of `val` downward along the `potential`, summing them as they reach common devices. If the trajectory down `potential` always leads to a single global minimum, then the trajectories form a spanning tree with the minimum at its root, and in the resulting field the root holds the sum of all values of `val`.

- `low-pass(alpha,val)`: This function takes a field `val` of real values, and at each device implements an exponential filter with blending constant `alpha`, thus acting as a low-pass filter smoothing rapid changes in the input `val` at each device.

Now consider an example of an application deployed into a museum, whose docents monitor their efficacy in part by tracking the number of patrons nearby while they are working. This application can be implemented by a simple function, taking as input Boolean fields indicating docents and patrons, and whose body is defined by purely-functional composition of the three blocks above, written e.g. in the following way:

```
def track-count(docent, patron) {
   low-pass( 0.5, converge-sum( distance-to(docent), mux(patron,1,0)))
}
```

in which the function `mux` acts as a simple multiplexer at each device, transforming true values to 1 and false values to 0. This function creates a field of estimated distances out of each `docent`, and uses it as potential field for counting the number of `patrons` nearby, with the low-pass filter smoothing the result so as to deal with rapid fluctuations in the estimate that can be caused by device mobility.

As the aim of this paper is to clarify syntax and semantics of the field-based computational model, this example should already clarify the goal of compositionally stacking increasingly complex distributed algorithms, up to a point in which the focus on individual agent behaviour completely vanishes. This can be taken even further by proving that the "building block" algorithms satisfy certain properties preserved by functional compositions, such as self-stabilisation [43] or consistency with a continuum model [7], thus implying the same properties hold for applications composed using those building blocks [5].

2.3 Higher-order fields and restriction

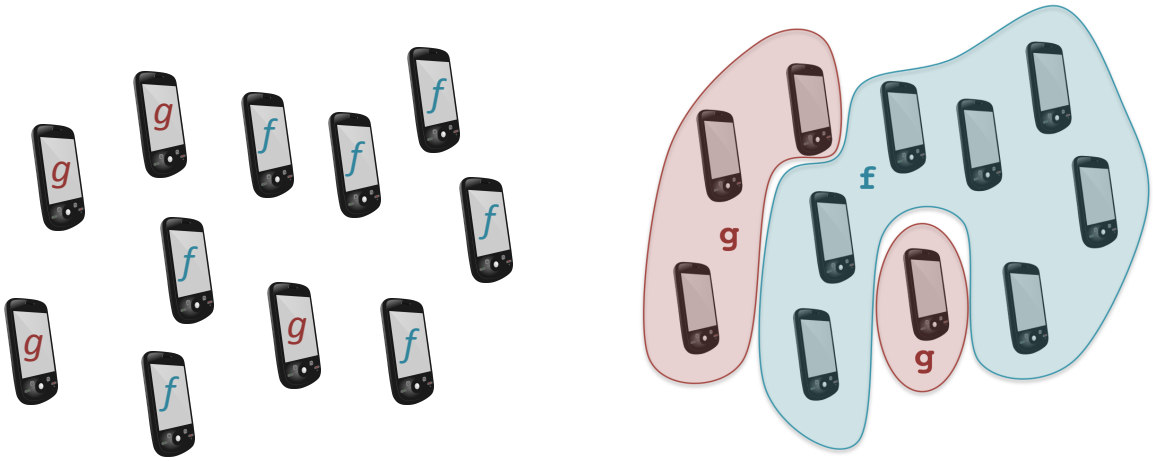
The calculus that we present in this paper is a higher-order extension of the work in [17] to include embedded first-class functions, with the primary goal of allowing field computations to handle functions just like any other value. This extension hence provides a number of advantages:

- Functions can take functions as arguments and return a function as result (higher-order functions). This is key to defining highly reusable building block functions, which can then be fully parameterised with various functional strategies.

- Functions can be created "on the fly" (anonymous functions). Among other applications, such functions can be passed into a system from the external environment, as a fields of functions considered as input coming a sensor modelling humans adding new code into a device while the system is operating.

- Functions can be moved between devices in the same way our calculus allows values to move, which allows one to express complex patterns of code deployment across space and time.

- Similarly, in our calculus a function value is naturally actually a field of functions (possibly created on the fly and then shared by movement to all devices), and can be used as an "aggregate function" operating over a whole spatial domain.



(a) A field of functions hosted on individual devices

(b) The same field interpreted as an aggregate function defined piecewise over aggregates of devices

Figure 1: Field calculus functions are evaluated over a domain of devices, e.g., in (a) the network reports function `f` in some devices, and function `g` in others. When such a field of functions is used in function application it gets reinterpreted as an aggregate function, as shown in (b). Namely, the domain is actually partitioned into two subspaces that do not interact with each other: in one domain D_f, function `f` is applied to its arguments, seen as fields restricted to D_f; in the complementary domain D_g, function `g` is applied to its arguments, seen as fields restricted to D_g.

The last feature is critical, and its implications are further illustrated in Figure 1. In considering fields of function values, we take the elegant approach in which making a function call acts as a branch, with each function in the range of the field applied only on the subspace of devices that hold that function. When the field of functions is constant, this implicit branch reduces to be precisely equivalent to a standard function call. This means that we can view ordinary evaluation of a function f as equivalent to creating a function-valued field with a constant value f, then making a function call applying that field to its argument fields. This elegant transformation is one of the key insight of this paper, enabling first-class functions to be implemented with relatively minimal complexity.

This interpretation of function calls as branching also turns out to be very flexible, as it generally allows the dynamic partitioning of the network into subspaces, each executing a different subprogram, and such programs can be even dynamically injected into some device and then be diffused around, as will be exemplified later in this paper.

3 The Higher-order Field Calculus: Syntax and Typing

This section presents the *higher-order field calculus (HFC)*, as extended and refined from [18],[2] a tiny functional calculus capturing the essential elements of field computations, much as λ-calculus [12] captures the essence of functional computation and FJ [26] the essence of object-oriented programming. For the key importance of higher-order features, especially in the toolchain under construction [40], in the following we sometimes refer to this calculus as simply the *field calculus*, especially when there is no confusion with the work in [17], which did not include higher-order features.

The defining property of fields is that they allow us to see a computation from two different viewpoints. On the one hand, from the standard "local" viewpoint, a computation is seen as occurring in a single device,

[2]The version of the HFC presented in this paper is a minor refinement of the version of HFC presented in [18]. The new version adopts a different syntax (in [18] a Lisp-like syntax was used), is parametric in the set of the modeled data values (in [18] Booleans, numbers, and pairs were explicitly modeled), and does not have a syntactic construct devoted to domain restriction (the `if`-construct in [18]) since it can be encoded by means of an aggregate function call—the existence of such an encoding is one of the novel contributions of this paper.

P	::=	$\overline{\mathtt{F}}\,\mathtt{e}$	program
F	::=	$\mathtt{def}\ \mathtt{d}(\overline{\mathtt{x}})\ \{\mathtt{e}\}$	function declaration
e	::=	$\mathtt{x} \mid \phi \mid \mathtt{c}(\overline{\mathtt{e}}) \mid \mathtt{b} \mid \mathtt{d} \mid (\overline{\mathtt{x}})\ \mathtt{=>}\ \mathtt{e} \mid \mathtt{e}(\overline{\mathtt{e}}) \mid \mathtt{rep}(\mathtt{e})\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}\} \mid \mathtt{nbr}\{\mathtt{e}\}$	expression

Figure 2: Syntax.

and it hence manipulates data values (e.g., numbers) and communicates such data values with other devices in order to enable coordination. On the other hand, from the "aggregate" (or "global") viewpoint [46], computation is seen as occurring on the overall network of interconnected devices: the data abstraction manipulated is hence a whole distributed *field*, a dynamically evolving data structure mapping devices δ in a domain D (the whole or a subset of it) to associated data values $\mathtt{v}$ in range V.[3] Field computations then take fields as input (e.g., from sensors) and produce new fields as outputs (e.g., to feed actuators). Both input and output may, of course, change over time (e.g., as inputs change or the computation progresses). For example, the input of a computation might be a field of temperatures, as perceived by sensors at each device in the network, and its output might be a Boolean field that maps to `True` where and when temperature is greater than 25°C, and to `False` elsewhere. In a more involved example, the output might map to `True` only those devices whose distance is less than 50 meters from some device where temperature was greater than 25°C for the last 60 seconds. The operational semantics described in Section 5 will then describe how such global-level behaviour can be turned into a fully-distributed computation.

3.1 Syntax

Figure 2 presents the syntax of the proposed calculus. Following [26], the overbar notation denotes metavariables over sequences and the empty sequence is denoted by $\bullet$. E.g., for expressions, we let $\overline{\mathtt{e}}$ range over sequences of expressions, written $\mathtt{e}_1, \mathtt{e}_2, \ldots \mathtt{e}_n$ $(n \geq 0)$.

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

Expressions `e` are the main entities of the calculus; terminologically, an expression can be: a variable `x`, used as function formal parameter; a *neighbouring field value* ϕ; a *data-expression* $\mathtt{c}(\overline{\mathtt{e}})$ (where `c` is a *data-constructor* with arity $m \geq 0$ and the m expressions $\overline{\mathtt{e}}$ are its arguments); a *built-in function* name `b`; a *declared function* name `d`; an *anonymous function* $(\overline{\mathtt{x}})$ `=>` `e` (where $\overline{\mathtt{x}}$ are the formal parameters and `e` is the body); a *function call* $\mathtt{e}(\overline{\mathtt{e}})$; a `rep`-*expression* $\mathtt{rep}(\mathtt{e}_0)\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}_1\}$, modelling time evolution; or an `nbr`-*expression* $\mathtt{nbr}\{\mathtt{e}\}$, modelling device-to-neighbourhood interaction. It should be noted that for the purpose of defining a foundational calculus, data-expressions and built-in representing purely functional operators could be dropped: we have have decided to include them since they simplify using the calculus to formalise non-trivial examples. Let the set of free variables in an expression `e` be denoted by $\mathbf{FV}(\mathtt{e})$, say that an expression `e` is *closed* if $\mathbf{FV}(\mathtt{e}) = \bullet$, and assume the main expression of any program must be closed. In section 3.3 we informally describe the meaning of these constructs, before the full formal treatment of denotational and operational semantics will given in the remainder of the paper.

3.2 Values

A key aspect of the calculus is that *an expression always models a whole field*. Let the denumerable set of *device identifiers* δ (which are unique numbers) be $\mathbf{D}$. A *firing event* ε (or simply event) is a point in space-time where a device δ "fires", namely, evaluates the main expression of the program. The outcome of the evaluation of a closed expression `e` at a given event gives a *value*, so we can select a (space-time) domain $D_{\mathtt{e}}$ and collect the values obtained in all events in that domain $D_{\mathtt{e}}$, and by doing so form a field over $D_{\mathtt{e}}$, which then represent a time-varying distributed structure.

[3] Note that this viewpoint can embrace both discrete domains (e.g., networks of devices) and continuous domains (e.g., the environment where computation acts upon); in this paper, however, we will restrict ourselves to treating fields with discrete domains. For a discussion of the relationship between computation on discrete and continuous domains, see [7]

$\mathtt{v}$	$::=$	$\ell \mid \phi$	value
ϕ	$::=$	$\overline{\delta} \mapsto \overline{\ell}$	neighbouring field value
ℓ	$::=$	$\mathtt{f} \mid \mathtt{c}(\overline{\ell})$	local value
$\mathtt{f}$	$::=$	$\mathtt{b} \mid \mathtt{d} \mid (\overline{\mathtt{x}})\ \mathtt{=>}\ \mathtt{e}$	function value

Figure 3: Values, neighbouring field values, local values, and function values for HFC.

The syntax of values is given in Figure 3. A value can be either a *local value* ℓ or a *neighbouring field value* ϕ. At a given event where a device δ fires, a local value represents an atomic data produced by δ, whereas a neighbouring field value is a map associating a local value ℓ to each neighbour of δ. Neighbouring field values are generally used to describe the outcome of some form of device-to-neighbour interaction as described below (sharing of values as of construct `nbr`, or sensing of local environment e.g., to estimate distances to neighbours). While neighbouring field values cannot be denoted in source programs, but only appear dynamically during computations, a local value ℓ can be denoted (it is in fact parts of the syntax) to represent a constant-valued field that maps each event to that value (e.g., value 1 represents a field always mapping each device to 1). Given that the calculus is higher-order, a local value can be:

- either a *data value* $\mathtt{c}(\ell_1, \cdots, \ell_m)$, consisting of a data-constructor `c` of arity $m \geq 0$ and $m \geq 0$ local value arguments—data values, simply written `c` when $m = 0$, can be Booleans `True` and `False`, numbers, strings, or structured values like pairs (e.g., $\mathtt{Pair}(3, \mathtt{Pair}(\mathtt{False}, 5))$) or lists (e.g., $\mathtt{Cons}(2, \mathtt{Cons}(4, \mathtt{Null}))$);
- or a *function value* `f`, consisting of either a built-in operator `b`, a declared function `d`, or a closed anonymous function expression $(\overline{\mathtt{x}})\ \mathtt{=>}\ \mathtt{e}$.

3.3 Informal semantics

While neighbouring field values, data-expressions and functions (built-in, declared function, and anonymous), trivially result in a constant field, the last three kinds of expression (`nbr`-expressions, `rep`-expressions, and function calls) represent the core field manipulation constructs provided by the calculus. Their value at each event may depend on both the particular device that is evaluating it, and also on the last event of its neighbours.

1. *Time evolution:* $\mathtt{rep}(\mathtt{e}_0)\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}\}$ is a "repeat" construct for dynamically changing fields, assuming device δ evaluates the application of its anonymous function $(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}$ repeatedly in asynchronous rounds. At the first round $(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}$ is applied to the value of $\mathtt{e}_0$ at δ, then at each step $(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}$ is applied to the value obtained at previous step. For instance, $\mathtt{rep}(0)\{(\mathtt{x})\ \mathtt{=>}\ +(\mathtt{x}, 1))\}$ counts how many rounds each device has computed.
2. *Neighbouring field construction:* $\mathtt{nbr}\{\mathtt{e}\}$ models device-to-neighbour interaction, by returning a field of neighbouring field values: each device is associated to value ϕ, which in turn maps any neighbour δ in the domain $D_{\mathtt{e}}$ to its most recent available value of `e` (e.g., obtained via periodic broadcast, as discussed in the operational semantics). Such neighbouring field values can then be manipulated and summarised with built-in operators. For instance, $\mathtt{min\text{-}hood}(\mathtt{nbr}\{\mathtt{e}\})$ maps each device to the minimum value of `e` across its neighbourhood.
3. *Function call:* $\mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$, where $n \geq 0$ and `e` evaluates to a field of function values. If the field is not constant, the application is evaluated separately in each *cluster*, i.e., subdomain of events where `e` evaluates to the same function value. Either way, we reduce to the case where the field obtained from `e` is a constant function `f` over a certain domain, and there can be two cases:
   - If `f` is a built-in operator `b`, $\mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ maps an event to the result of applying `b` to the values at the same event of its $n \geq 0$ arguments $\mathtt{e}_1, \ldots, \mathtt{e}_n$. Note that `b` can be a *pure operator*, involving neither state nor communication (e.g. mathematical functions like addition, comparison, and sine)—for instance, $+(1,2)$ is the expression evaluating to the constant-valued field 3, also written $1+2$ for readability as for any other binary built-in operator. Alternatively, `b` can be an

environment-dependent operator, modelling a sensor—for instance, 0-ary `sns-temp` is used to map each device δ where the built-in operator call is evaluated to its local value of temperature, and the 0-ary `nbr-range` operator returns a neighbouring field value mapping each neighbour of the device δ where the built-in operator call is evaluated to an estimate of its current distance from δ.

- If `f` is not a built-in operator, it can be a declared function `d` with corresponding declaration `def d(x₁ … xₙ) {e}`, or an anonymous function `(x₁ … xₙ) => e`; then, expression $\mathtt{e}(\mathtt{e}_1,\ldots,\mathtt{e}_n)$ maps an event to the result of evaluating the closed expression $\mathtt{e}_0$ obtained from the body `e` of the function `f` by replacing the occurrences of the formal parameters $\mathtt{x}_1,\ldots,\mathtt{x}_n$ with the values of the expressions $\mathtt{e}_1,\ldots,\mathtt{e}_n$.

Remark 1 (Function Equality)**.** *In the definition of function calls given above, it is necessary to specify when two functional values* `f` *are "the same". In the remainder of this paper, we assume that two such values are the same when they are* syntactically equal*. This convention is carried over in the meaning of the builtin operator* $=$ *when applied to function values, so that* $=(\mathtt{f}_1,\mathtt{f}_2)$ *with* $\mathtt{f}_1$, $\mathtt{f}_2$ *values holds precisely when* $\mathtt{f}_1$, $\mathtt{f}_2$ *are syntactically identical expressions.*

According to the explanation given above, calling a declared or anonymous function acts as a branch, with each function in the range applied only on the subspace of devices that hold that function. Moreover, functional values allow code to be dynamically injected, moved, and executed in network (sub)domains. Namely: *(i)* functions can take functions as arguments and return a function as result (higher-order functions); *(ii)* (anonymous) functions can be created "on the fly"; *(iii)* functions can be moved between devices (via the `nbr` construct); and *(iv)* the function one executes can change over time (via the `rep` construct).

In this section, we have described the various constructs working in isolation: more involved examples dealing with combinations of constructs will be given in later sections, when the denotational and operational semantics are discussed.

Remark 2 (Syntactic Sugar)**.** *Conventional branching can be implemented using a function call: in the examples we give, we will use the* `if`*-expression* $\mathtt{if}\ (\mathtt{e}_0)\ \{\mathtt{e}_1\}\ \mathtt{else}\ \{\mathtt{e}_2\}$ *as syntactic sugar for:*

$$\mathtt{mux}(\mathtt{e}_0,()\mathtt{=>}\mathtt{snd}(\mathtt{Pair}(\mathtt{True},\mathtt{e}_1)),()\mathtt{=>}\mathtt{snd}(\mathtt{Pair}(\mathtt{False},\mathtt{e}_2)))()$$

where function `mux` *is a built-in function multiplexer as defined in Section 2.2, and function* `snd` *extracts the second component of a pair. The result of branching is to partition the space-time domain in two subdomains: in the one where* $\mathtt{e}_0$ *evaluates to* `True`*, field* $\mathtt{e}_1$ *is computed and returned; in the complement where* $\mathtt{e}_0$ *evaluates to* `False` *, field* $\mathtt{e}_2$ *is computed and returned.*

3.4 Typing

In this section, we present a variant of the Hindley-Milner type system [16] for the proposed calculus. This type system has two main kinds of types, *local types* (the types for local values) and *field types* (the types for neighbouring field values), and is designed specifically to ground the denotational semantics presented in next section and to guarantee the following two properties:

Type Preservation For every well-typed closed expression `e` of type T, if the evaluation of `e` on event ε (cf. the explanation at the beginning of Section 3.2) yields a result `v`, then `v` is of type T.

Domain Alignment For every well-typed closed expression `e` of type T, if the evaluation of `e` at event ε yields a neighbouring field value ϕ, then the domain of ϕ consists of the device δ of event ε and of its aligned neighbours, that is, the neighbours that have calculated the same expression `e` before the current evaluation started.

Domain alignment is key to guarantee that the semantics correctly relates the behaviour of `nbr`, `rep` and function application—namely, two neighbouring field values with different domain are never allowed to be combined. In essence, domain alignment is required in order to provide lexical scoping: without it, information may leak unexpectedly between devices that are evaluating different functions, or may be blocked from passing between devices evaluating the same function.

Since the type system is a customisation of the Hindley-Milner type system [16] to the field calculus, 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. This algorithm is based on an *unification* routine as in [32] which exists since the type variables t, l, r, s form a boolean algebra (i.e. s is exactly the intersection of l and r, while t is their union).

The syntax of types and local type schemes is given in Figure 4 (top). A *type* T is either a *type variable* t, or a local type, or a field type. A *local type* L is either a *local type variable* l, or a built-in type B (numbers, booleans, pairs, lists etc.), or the type of a function $(\overline{T}) \to R$ (possibly of arity zero). Note that a function always has local type, regardless of the local or field type of its arguments. A *return type* R is either a *return type variable* r, or a local return type, or a field type. A *local return type* S is either a *local return type variable* s, or a built-in type B, or the type of a function $(\overline{T}) \to S$ (possibly of arity zero). A *field type* F is the type $\texttt{field}(S)$ of a field whose range contains values of local return type S. Notice that the type system does not contemplate types of the kind $(\ldots) \to \cdots \to (\ldots) \to F$ (functions that return functions that return neighbouring field values), since expressions involving such types can be unsafe (as exemplified in the next subsection). Notice also that L is equal to S together with functions $(\overline{T}) \to F$ (thus T is R together with such functions).

Local type schemes, ranged over by LS, support typing polymorphic uses of data constructors, built-in operators and user defined-functions. Namely, for each data constructor, built-in operator or user-defined function g there is a *local type scheme* $\forall \overline{t}\overline{l}\overline{r}\overline{s}.L$, where $\overline{t}$, $\overline{l}$, $\overline{r}$ and $\overline{s}$ are all the type variables occurring in the type L, respectively. Each use of g can be typed with any type obtained from $\forall \overline{t}\overline{l}\overline{r}\overline{s}.L$ by replacing the type variables $\overline{t}$ with types, $\overline{l}$ with local types, $\overline{r}$ with return types and $\overline{s}$ with local return types.

Type environments, ranged over by $\mathscr{A}$ and written $\overline{\mathtt{x}} : \overline{T}$, are used to collect type assumptions for program variables (i.e., the formal parameters of the functions and the variables introduced by the rep-construct). *Local-type-scheme environments*, ranged over by $\mathscr{D}$ and written $\overline{\mathtt{g}} : \overline{LS}$, are used to collect the local type schemes for the data constructors and built-in operators together with the local type schemes inferred for the user-defined functions. In particular, the distinguished *built-in local-type-scheme environment* $\mathscr{B}$ associates a local type scheme to each data constructor c and built-in function b — Figure 5 shows the local type schemes for the data constructors and built-in functions used throughout this paper. We distinguish the built-in functions in *pure* (their evaluation only depends on arguments) and *non-pure* (their evaluation can depend on the specific device and on its physical environment, like e.g. for sensors).

We use the convention that if b is a built-in unary operator with local argument and return type, b[f] denote the corresponding operator on neighbouring field values which apply b pointwise to its argument: in other words, $\texttt{b[f]}(\phi)$, which is equivalent to $\texttt{map-hood}(\texttt{b}, \phi)$ (see Figure 5), at any device maps a neighbour δ to the result of applying b to the value of ϕ at δ. If b is a multi-ary operator, a notation such as b[f,l] or b[l,f,f] is used to specify which parameters have to be promoted to neighbouring field values: for instance, at each device, $\texttt{+[f,f]}(\phi_1, \phi_2)$ gives a neighbouring field value mapping a neighbour δ to the sum of values of ϕ_1 and ϕ_2 at δ. Notice that the definition of built-in operator map-hood is actually a schema defining such an operator for any positive ariety of its arguments $\overline{l}$.

The type rules are given in Figure 4 (bottom). The typing judgement for expressions is of the form "$\mathscr{D}; \mathscr{A} \vdash \mathtt{e} : T$", to be read: "e has type T under the local-type-scheme assumptions $\mathscr{D}$ (for data constructors, built-in operators and user-defined functions) and the type assumptions $\mathscr{A}$ (for the program variables occurring in e)". As a standard syntax in type systems [26], given $\overline{T} = T_1, \ldots, T_n$ and $\overline{\mathtt{e}} = \mathtt{e}_1, \ldots, \mathtt{e}_n$ ($n \geq 0$), we write $\mathscr{D}; \mathscr{A} \vdash \overline{\mathtt{e}} : \overline{T}$ as short for $\mathscr{D}; \mathscr{A} \vdash \mathtt{e}_1 : T_1 \cdots \mathscr{D}; \mathscr{A} \vdash \mathtt{e}_n : T_n$. Note that the type rules are syntax-directed, so they straightforwardly describe a type inference algorithm.

Rule [T-VAR] (for variables) lookups the type assumptions for x in $\mathscr{A}$.

Rule [T-VAL] (for values) lookups the specifications of the data constructor c in $\mathscr{B}$ and checks that are met by its arguments, which need to be *values*. This request allows us to recognize whether a well-typed expression is a value by only checking its outermost syntactic block, which is convenient for later proofs. For convenience of presentation, in the following we shall assume that every constructor $\mathtt{c} : (\overline{L}) \to S$ comes with an associated function $\mathtt{c}'$ defined as $(\overline{\mathtt{x}})$ => $\mathtt{c}(\overline{\mathtt{x}})$. Notice that specifications for constructors are only allowed to involve local types.

Types:

$$
\begin{array}{lcl|l|l}
T & ::= & t & F & L \\
L & ::= & l & B & (\overline{T}) \to R \\
R & ::= & r & F & S \\
S & ::= & s & B & (\overline{T}) \to S \\
F & ::= & \multicolumn{3}{l}{\texttt{field}(S)}
\end{array}
\qquad
\begin{array}{r}
\text{type} \\ \text{local type} \\ \text{return type} \\ \text{local return type} \\ \text{field type}
\end{array}
$$

Local type schemes:

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

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

$$\frac{\text{[T-VAR]}}{\mathscr{D};\mathscr{A},\texttt{x}:T \vdash \texttt{x}:T} \qquad \frac{\text{[T-VAL]} \quad \mathscr{B} \vdash \texttt{c} : (\overline{L}) \to S \qquad \mathscr{D};\mathscr{A} \vdash \overline{\ell} : \overline{L}}{\mathscr{D};\mathscr{A} \vdash \texttt{c}(\overline{\ell}) : S}$$

$$\frac{\text{[T-N-FUN]} \quad L' = L\bar{t} := \overline{T}\bar{l} := \overline{L}\bar{r} := \overline{R}\bar{s} := \overline{S}}{\mathscr{D},\texttt{g} : \forall \bar{t}\bar{l}\bar{r}\bar{s}.L;\mathscr{A} \vdash \texttt{g} : L'}$$

$$\frac{\text{[T-A-FUN]} \quad \overline{\texttt{y}} = \mathbf{FV}((\overline{\texttt{x}})\texttt{=>e}) \quad \mathscr{A}(\overline{\texttt{y}}) \text{ local types} \quad \mathscr{D};\ \mathscr{A},\overline{\texttt{x}}:\overline{T} \vdash \texttt{e} : R}{\mathscr{D};\mathscr{A} \vdash (\overline{\texttt{x}})\texttt{=>e} : (\overline{T}) \to R}$$

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

$$\frac{\text{[T-REP]} \quad \mathscr{D};\mathscr{A} \vdash \texttt{e}_1 : S \qquad \mathscr{D};\mathscr{A},\texttt{x}:L \vdash \texttt{e}_2 : S}{\mathscr{D};\mathscr{A} \vdash \texttt{rep}(\texttt{e}_1)\{(\texttt{x})\texttt{=>e}_2\} : S} \qquad \frac{\text{[T-NBR]} \quad \mathscr{D};\mathscr{A} \vdash \texttt{e} : S}{\mathscr{D};\mathscr{A} \vdash \texttt{nbr}\{\texttt{e}\} : \texttt{field}(S)}$$

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

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

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

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

Figure 4: HFC: types, local type schemes and type rules for expressions, function declarations, and programs.

Rule [T-N-FUN] (for built-in functions and user-defined function names) ensures that the local type scheme $\forall \bar{t}\bar{l}\bar{r}\bar{s}.L$ associated to the built-in function or user-defined function name g is instantiated by substituting the type variables $\bar{t}$, $\bar{l}$, $\bar{r}$, $\bar{s}$ correctly with types in T, L, R, S respectively.

Rule [T-A-FUN] (for anonymous functions) ensures that anonymous functions $(\overline{\texttt{x}})$=>e have return type in R and do not contain free variables of field type in order to avoid domain alignment errors (as exemplified in the next subsection).

Rule [T-APP] (for function applications) is standard.

Rule [T-REP] (for rep-expressions) ensures that both the variable x, its initial value $\texttt{e}_1$ and the body $\texttt{e}_2$ have (the same) local return type. In fact, allowing field types might produce domain mismatches, while a rep-expression of type $(\overline{T}) \to F$ would be a non-constant (thus not safely applicable) function returning neighbouring field values.

Rule [T-NBR] (for nbr-expressions) ensures that the body e of the expression has a local return type. This prevents the attempt to create a "field of fields" (i.e., a neighbouring field value that maps device identities to neighbouring field values).

Function declaration typing (represented by judgement "$\mathscr{D} \vdash \texttt{F} : LS$") and program typing (represented by judgement "$\vdash \texttt{P} : L$") are almost standard.

We say that a program P is *well-typed* to mean that $\vdash \texttt{P} : L$ holds for some local type L.

Remark 3 (On Termination)**.** *Termination of a device firing is clearly not decidable. In the rest of the paper*

Built-in constructors:

$$
\begin{aligned}
\mathscr{B}(\texttt{True}) &= () \to \texttt{bool} \\
\mathscr{B}(\texttt{False}) &= () \to \texttt{bool} \\
\mathscr{B}(0) &= () \to \texttt{num} \\
\mathscr{B}(\texttt{Pair}) &= \forall s_1 s_2.(s_1, s_2) \to \texttt{pair}(s_1, s_2) \\
\mathscr{B}(\texttt{Null}) &= \forall s.() \to \texttt{list}(s) \\
\mathscr{B}(\texttt{Cons}) &= \forall s.(s, \texttt{list}(s)) \to \texttt{list}(s)
\end{aligned}
$$

Pure built-in functions (independent from the current device and value-tree environment):

$$
\begin{aligned}
\mathscr{B}(\texttt{pair[f,f]}) &= \forall s_1 s_2.(\texttt{field}(s_1), \texttt{field}(s_2)) \to \texttt{field}(\texttt{pair}(s_1, s_2)) \\
\mathscr{B}(\texttt{pair[l,f]}) &= \forall s_1 s_2.(s_1, \texttt{field}(s_2)) \to \texttt{field}(\texttt{pair}(s_1, s_2)) \\
\mathscr{B}(\texttt{fst}) &= \forall s_1 s_2.(\texttt{pair}(s_1, s_2)) \to s_1 \\
\mathscr{B}(\texttt{snd}) &= \forall s_1 s_2.(\texttt{pair}(s_1, s_2)) \to s_2 \\
\mathscr{B}(\texttt{head}) &= \forall s.(\texttt{list}(s)) \to s \\
\mathscr{B}(\texttt{tail}) &= \forall s.(\texttt{list}(s)) \to \texttt{list}(s) \\
\mathscr{B}(\texttt{min-hood}) &= \forall s.(\texttt{field}(s)) \to s \\
\mathscr{B}(\texttt{min-hood+}) &= \forall s.(\texttt{field}(s)) \to s \\
\mathscr{B}(\texttt{pick-hood}) &= \forall s.(\texttt{field}(s)) \to s \\
\mathscr{B}(\texttt{map-hood}) &= \forall \overline{s}.\forall s'((\overline{s}) \to s', \texttt{field}(\overline{s})) \to \texttt{field}(s') \\
\mathscr{B}(\texttt{fold-hood}) &= \forall s.((s, s) \to s, \texttt{field}(s)) \to s \\
\mathscr{B}(\texttt{mux}) &= \forall s.(\texttt{bool}, s, s) \to s \\
\mathscr{B}(\texttt{mux[f,f,l]}) &= \forall s.(\texttt{field}(\texttt{bool}), \texttt{field}(s), s) \to \texttt{field}(s) \\
\mathscr{B}(\texttt{and}) &= (\texttt{bool}, \texttt{bool}) \to \texttt{bool} \\
\mathscr{B}(\texttt{*}) &= (\texttt{num}, \texttt{num}) \to \texttt{num} \\
\mathscr{B}(\texttt{-}) &= (\texttt{num}, \texttt{num}) \to \texttt{num} \\
\mathscr{B}(\texttt{+}) &= (\texttt{num}, \texttt{num}) \to \texttt{num} \\
\mathscr{B}(\texttt{+[f,f]}) &= (\texttt{field}(\texttt{num}), \texttt{field}(\texttt{num})) \to \texttt{field}(\texttt{num}) \\
\mathscr{B}(\texttt{<[f,l]}) &= \forall s.(\texttt{field}(s), s) \to \texttt{field}(\texttt{bool}) \\
\mathscr{B}(\texttt{=}) &= \forall t.(t, t) \to \texttt{bool}
\end{aligned}
$$

Non-pure built-in functions (depend from the current device and value-tree environment):

$$
\begin{aligned}
\mathscr{B}(\texttt{sns-range}) &= () \to \texttt{num} \\
\mathscr{B}(\texttt{sns-injection-point}) &= () \to \texttt{bool} \\
\mathscr{B}(\texttt{sns-injected-function}) &= () \to (() \to \texttt{num}) \\
\mathscr{B}(\texttt{nbr-range}) &= () \to \texttt{field}(\texttt{num}) \\
\mathscr{B}(\texttt{uid}) &= () \to \texttt{num}
\end{aligned}
$$

Figure 5: Local type schemes for the built-in functions used throughout this paper.

we assume without loss of generality for the results of this paper that a decidable subset of the termination fragment has been identified by applying some static analysis technique for termination.

3.5 Examples

Though the type system mostly trivially assigns a local or field type to an expression, it enforces some peculiar restrictions that we now clarify with the help of few examples. In the code to come, syntax is coloured to increase readability: grey for comments, red for field calculus keywords, and blue for functions (both user defined and built in). In the first-order type system presented in [46] one peculiar check was introduced in the type system: a conditional `if` expression could not have field type. In fact, such an expression produces a field combining two subfields where values, which are neighbouring field values, are restricted in different ways, each to the neighbours that evaluated the conditional guard in the same way. Such a combination, hence, is shown to contradict domain alignment as described by the following example—in the present language conditional branching is modeled by the branching implicit in function application (see Remark 2), thus similar issues apply to functions returning neighbouring field values. Consider the expression $\texttt{e}_{\texttt{wrong}}$:

```
(if (uid=1) {(x)=>x} else {(x)=>x +[f,f] nbr{uid}} )(nbr{0}) +[f,f] nbr{uid}
```

This expression violates domain alignment, thus provoking conflicts between field domains. When the conditional expression is evaluated on a device with `uid` (unique identifier) equals to 1, function $\texttt{f} = \texttt{(x)=>x}$ is obtained whence applied to $\texttt{nbr}\{0\}$ *in the restricted domain of devices who computed the same* `f` *in their last evaluation round*, that is the domain $\{1\}$. Thus at device 1 the function application returns the neighbouring field value $\phi = 1 \mapsto 0$ which cannot be combined with $\texttt{nbr}\{\texttt{uid}\}$ whose (larger) domain consists of *all* neighbours of device 1. A complementary violation occurs for all neighbors of device 1, which compute a neighbouring field value whose domain lacks device 1.

However, not all expressions involving functions returning fields are unsafe. For instance, consider the similar expression $\texttt{e}_{\texttt{safe}}$:

```
((x)=>x)(nbr{0}) +[f,f] nbr{uid}
```

In this case, on every device the same function $\texttt{f} = \texttt{(x)=>x}$ is obtained whence applied to $\texttt{nbr}\{0\}$, that is, no alignment is required thus the function application returns the whole neighbouring field value $\phi = \texttt{nbr}\{0\}$ which can be safely combined with $\texttt{nbr}\{\texttt{uid}\}$. This suggest that functions returning neighbouring field values are safe *as long as they evaluate to the same function regardless of the device and surrounding environment*.

The type system presented in the previous Section 3.4 ensures this distinction. Besides performing standard checks, the type system perform the following additional checks in order to ensure domain alignment:

- *Functions returning neighbouring field values are not allowed as return type*. That is, all functions (both user defined and built-ins, and as a consequence also `rep` statements) don't return a "function returning neighbouring field values". This prevents the possibility of having a well-typed expression `e` which evaluates to different functions returning neighbouring field values on different devices, thus allowing undesired behaviours such as in the example described above. In fact, if we expand the conditional in $\texttt{e}_{\texttt{wrong}}$ according to Remark 2, we obtain

  ```
  mux(uid=1, ()=>snd(Pair(True,(x)=>x)),
             ()=>snd(Pair(False,(x)=>x+[f,f]nbr{uid})))(...)
  ```

  in which both the entire `mux` expression and both of its two branches have the disallowed type $() \to \texttt{field}(\texttt{num}) \to \texttt{field}(\texttt{num})$.

- *In an anonymous function* $(\overline{\texttt{x}})\texttt{=>e}$*, the free variables* $\overline{\texttt{y}}$ *of* `e` *that are not in* $\overline{\texttt{x}}$ *have local type*. This prevents a device δ from creating a closure $\texttt{e}' = (\overline{\texttt{x}})\texttt{=>e}[\overline{\texttt{y}} := \overline{\phi}]$ containing neighbouring field values $\overline{\phi}$ (whose domain is by construction equal to the subset of the aligned neighbours of δ). The closure $\texttt{e}'$ may lead to a domain alignment error since it may be shipped (via the `nbr` construct) to another device δ' that may use it (i.e., apply $\texttt{e}'$ to some arguments); and the evaluation of the body of $\texttt{e}'$ may involve use of a neighbouring field value ϕ in $\overline{\phi}$ such that the set of aligned neighbours of δ' is different from the domain of ϕ. For instance, the expression $\texttt{e}'_{\texttt{wrong}}$:

  ```
  ((x) => pick-hood(nbr{() => min-hood(x +[f,f] nbr{0})}))(nbr{0})()
  ```

  (where `pick-hood` is a built-in function that returns the value of a randomly chosen device among a device neighbours) that should have type `num`, is ill-typed. Its body will fail to type-check since it contains the function $()\texttt{=>min-hood}(\texttt{x} + \texttt{nbr}\{0\})$ with free variable `x` of field type. This prevents conflicts between field domains since:

  - when the expression is evaluated on a device δ, the closure

    $$\ell = \texttt{() => min-hood(x +[f,f] nbr}\{0\}\texttt{)}[\texttt{x} := \phi_2]$$

where ϕ_2 is the neighbouring field value produced by the evaluation of `nbr{0}` on δ (whose domain consists of the aligned neighbours of the device δ—i.e., the neighbours that have evaluated a corresponding occurrence of $\mathtt{e}'_{\mathtt{wrong}}$ in their last evaluation round), will be made available to other devices; and

– when the expression is evaluated on a device δ' that has δ as neighbour and the evaluation of the application of `pick-hood` returns the closure ℓ received from δ; then the neighbouring field value ϕ'_1 produced by the evaluation of the subexpression `nbr{0}` of ℓ on δ' would contain the aligned neighbours of the device δ' (i.e., the neighbours that have evaluated a corresponding occurrence of $\mathtt{e}'_{\mathtt{wrong}}$ in their last evaluation round) hence may have a domain different from the domain of ϕ_2, leaving the neighbouring field values mismatched in domain at the evaluation of the sum occurring in ℓ on δ'.

- *In a* `rep`*-expression* $\mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\mathtt{=>}\mathtt{e}_2\}$ *it holds that* `x`, $\mathtt{e}_1$ *and* $\mathtt{e}_2$ *have (the same) local return type*. This prevents a device δ from storing in `x` a neighbouring field value ϕ that may be reused in the next computation round of δ, when the set of the set of aligned neighbours may be different from the domain of ϕ. For instance, the expression $\mathtt{e}''_{\mathtt{wrong}}$:

```
min-hood(rep(nbr{0}){(x) => x +[f,f] nbr{uid}})
```

  that should have type `num`, is ill-typed.

- *In a* `nbr`*-expression* `nbr{e}` *the expression* `e` *has local type*. This prevents the attempt to create a "field of fields" (i.e., a neighbouring field value that maps device identifiers to neighbouring field values)—which is pragmatically often overly costly to maintain and communicate, as well as further complicating the issues involved in ensuring domain alignment.

4 Denotational Semantics

We now introduce a denotational semantics for the field calculus. In this semantics, we are posed with an additional challenge with respect to the denotational semantics for lambda calculus or ML-like programming languages (see among many [38, 32, 50]): several devices and firing events are involved in the computation, possibly influencing each other's outcomes. Even though this scenario seems similar to classical concurrent programming (see e.g. [1, 19]), it cannot be treated using the same tools because of two crucial differences:

- communication within a computational round is strongly connected with the underlying space-time properties of the physical world in which devices are located;

- the whole outcome of the computation is not a single value, but a *field* of spatially and temporally distributed values.

In order to reflect these characteristics of field calculus, the denotational semantics of an expression is given in terms of the resulting space-time field, formalised as a partial mapping from the "evolving" domain (the set of participating devices may change over time) to values. The domain is defined as a set of (firing) events, each carrying a node identifier and equipped with a neighbour relationship modelling causality, i.e., reachability of communications. Values are defined analogously to ML-like languages, with the addition of fields and formulating functions as mathematical operators on computational fields instead of single values (which is necessary because the computation of a function might involve state communication among devices or events).

This semantics is nicely compositional, allowing one to formalise each construct separately. Furthermore, it represents the global result of computation as a single "space-time object", thus giving a perspective that is more proper for the designer of a computation and more convenient in proving certain properties of the calculus—the operational semantics (Section 5) is instead more useful in designing a platform for the equivalent distributed execution of field computations on actual devices.

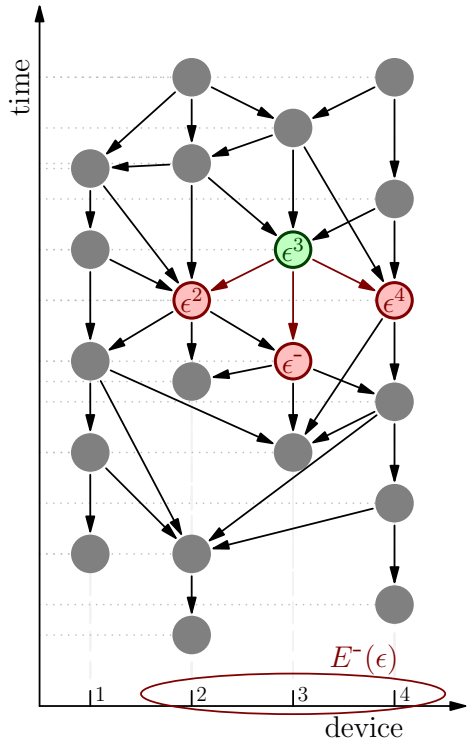


Figure 6: A sample neighbourhood graph on four asynchronously firing devices. The neighbours of the green event $\varepsilon = \varepsilon^3$ are shown in red, forming a neighbourhood over devices 2, 3, and 4.

4.1 Preliminary definitions

Recall that we let **D** be the set of *devices*, ranged over by meta-variable δ; we now also let **E** be the set of *events*, ranged over by meta-variable ε. An event models a firing in a network, and is labeled by a device identifier δ_ε. We use E to range over subsets of **E**.

We model the neighbour relationship as a global-level, fixed predicate $neigh(\varepsilon, \varepsilon')$ which holds if the device at ε is aware of the result of computation at ε'. This relationship is based on the topology and time evolution of involved devices, hence we require that $neigh(\varepsilon, \varepsilon')$ satisfies the following properties:

1. the graph on **E** induced by *neigh* is a DAG (directed acyclic graph);
2. every ε' linked from ε has a different label $\delta_{\varepsilon'}$, that is, there exists no ε', ε'' such that $\delta_{\varepsilon'} = \delta_{\varepsilon''}$ and both $neigh(\varepsilon, \varepsilon')$ and $neigh(\varepsilon, \varepsilon'')$ hold;
3. every ε' is linked from at most one ε with the same label $\delta_\varepsilon = \delta_{\varepsilon'}$, that is, there exists no ε_0, ε_1 such that $\delta_{\varepsilon_0} = \delta_{\varepsilon_1} = \delta_{\varepsilon'}$ and both $neigh(\varepsilon_0, \varepsilon')$ and $neigh(\varepsilon_1, \varepsilon')$ hold;

Property 1 ensures that *neigh* is causality-driven, property 2 that the neighbours of an event ε are indexed by devices, and property 3 that restricting *neigh* to a single device we obtain a set of directed paths (thus modeling that the firing of a device is aware of either its immediate predecessor on the same device or nothing).

Figure 6 shows a sample neighbourhood graph involving four devices, each firing from four to six times. Notice that device 2 is restarted after its second firing, and that the set of neighbouring devices changes over time for each device, in particular, device 4 drops its connection with device 2 from its fourth firing on. This can be explained by assuming devices to be moving in space—though movement is not represented in Figure 6. Nonetheless, the depicted graph satisfies all of the three above mentioned properties.

For all ε and δ, we define ε^δ as the latest event at δ that ε can be aware of, namely the one satisfying $neigh(\varepsilon, \varepsilon^\delta)$ if $\delta \neq \delta_\varepsilon$ or ε itself in case $\delta = \delta_\varepsilon$. Notice that if ε^δ exists, it is unique by property 2. We use ε^- to denote the previous event of ε at the same device *if it exists*. These notations are exemplified in the picture above, where $\varepsilon = \varepsilon^3$. We also define $E^-(\varepsilon)$ where $E \subseteq \mathbf{E}$ as the neighbourhood of E, namely, the

set of devices δ such that ε^{δ} exists in E. For example, $E^{-}(\varepsilon) = \{2,3,4\}$ for the green event ε in the picture above.

We chose such a generic approach to model neighbouring in order to abstract from the particular conditions and implementations that might occur in practice when an execution platform has to handle device to device communication.

Example 1 (Unit-Disc Communication)**.** *A typical scenario for the computations we aim at modelling and designing is that of a mobile set of wirelessly communicating devices, such that the neighbourhood relationship depends primarily on physical position—e.g., devices within a certain range can communicate. In this case, the predicate $neigh(\varepsilon,\varepsilon')$ could be defined from a set of* paths $\mathbf{P}$ *for moving devices, labels t_ε modelling passage of time, a timeout value $t_{\mathtt{d}}$, and a predicate $neigh(p,p')$ between positions, where:*

- *$neigh(p,p')$ is a global-level, fixed reflexive and symmetric predicate which holds if the two devices at positions p and p' are neighbours.*
- *$\mathbf{P}$ is a mapping from device identifiers δ to space-time paths P. A path P is a continuous function from $\mathbb{R}^+$ (times) to the set of possible positions, defined on the union of a finite number of disjoint closed intervals (the time intervals in which the device is turned on).*[4]
- *$t_{\mathtt{d}} \in \mathbb{R}^+$ models a timeout expiration after which non-communicating devices are considered "removed," allowing adaptation of the network to device removal and topology changes.*
- *$neigh(\varepsilon,\varepsilon')$ holds if and only if:*
  1. *$t_{\varepsilon'} \in [t_\varepsilon - t_{\mathtt{d}}, t_\varepsilon)$ (i.e. ε' happened in the time interval of size $t_{\mathtt{d}}$ before ε);*
  2. *$\mathbf{P}(\delta_\varepsilon)$ is defined in the interval $[t_{\varepsilon'}, t_\varepsilon]$ (i.e. δ_ε was constantly turned on during the time between events ε' and ε);*
  3. *$neigh(\mathbf{P}(\delta_\varepsilon)(t_{\varepsilon'}), \mathbf{P}(\delta_{\varepsilon'})(t_{\varepsilon'}))$ holds (i.e. the two devices where neighbours when ε' happened);*
  4. *there exists no further event ε'' with $\delta_{\varepsilon''} = \delta_{\varepsilon'}$ and $t_{\varepsilon''} > t_{\varepsilon'}$ satisfying the above conditions (i.e. ε' is the last firing of $\delta_{\varepsilon'}$ recorded by δ_ε before ε).*

4.2 Denotational semantics of types

A necessary preliminary step in the definition of denotational semantics for the field calculus is to clarify the denotation of types. As usual, the denotation of a type gives a set over which the denotation of expressions that are given that type range—denotation of expressions will be presented in next section. The denotational semantics of a type is given by two intertwined functions: a function $\mathscr{V}[\![\cdot]\!]$ mapping a type T without type variables to a set of local value denotations (i.e. values at individual devices), and a function $\mathscr{T}[\![\cdot]\!]$ mapping T to a set of *field evolutions*, ranged over by meta-variable Φ, assigning local values to every device in each firing event.

If B is a built-in local type, we assume that $\mathscr{V}[\![B]\!]$ is given. For derived types, $\mathscr{V}[\![\cdot]\!]$ and $\mathscr{T}[\![\cdot]\!]$ are altogether defined by rules:

$$\begin{array}{rcl} \mathscr{T}[\![T]\!] & = & \mathbf{E} \nrightarrow \mathscr{V}[\![T]\!] \\ \mathscr{V}[\![\mathtt{field}(L)]\!] & = & \mathbf{D} \nrightarrow \mathscr{V}[\![L]\!] \\ \mathscr{V}[\![(T_1,\ldots,T_n) \to T]\!] & = & \mathbf{F} \times (\mathscr{T}[\![T_1]\!] \times \ldots \times \mathscr{T}[\![T_n]\!]) \to \mathscr{T}[\![T]\!] \end{array}$$

where $\mathbf{E} \nrightarrow \mathscr{V}[\![T]\!]$ (resp. $\mathbf{D} \nrightarrow \mathscr{V}[\![T]\!]$) is the set of partial functions from $\mathbf{E}$ (resp. $\mathbf{D}$) to $\mathscr{V}[\![T]\!]$, and $\mathbf{F}$ is a set of function tags uniquely characterizing each function.

The denotation of a type T is a set of field evolutions, that is, partial maps from events $\mathbf{E}$ to local value denotations $\mathscr{V}[\![T]\!]$. This reflects the fact that an expression $\mathtt{e}$ evaluates to (possibly different) local values in each device and event of the computation.

[4] We remark that this definition for paths allows (in an extension of the present language) consideration of devices in which some stored values are preserved while turned off.

The local value denotation of a field type $\texttt{field}(L)$ is the set of partial functions from devices to local value denotations, which are intended to map a neighbourhood (or an "aligned subset" of it: in both cases, a subset of $\mathbf{D}$) to local value denotations of the corresponding local type.

The denotation of a function $(T_1, \ldots, T_n) \to T$ is instead a set of pairs with the following two components:

- The function tag in $\mathbf{F}$ (e.g. a syntactic function value as in Figure 3), needed in order to reflect the choice to compare functions by *syntactic* equality instead of *semantic* equality, which would not allow a computable operational semantics (see Remark 1). In fact, the presence of such tags is used to grant that two differently specified but identically behaving functions $\texttt{f}, \texttt{f}'$ get distinct denotations.
- A mapping from input field evolutions in $\mathscr{T}[\![T_1]\!], \ldots \mathscr{T}[\![T_n]\!]$ to an output field evolution in $\mathscr{T}[\![T]\!]$.

The local execution environment under which the computation of the function is assumed to happen is implicitly determined as the (common) domain of its input field evolutions; and the same domain will be retained for the output. This environment can influence the outcome of the computation through `rep` and `nbr` statements and through non-pure built-in functions.

Since local denotational values are not connected to specific events or domains, the common domain of the input field evolutions can be any subset of $\mathbf{E}$. In particular, this fact implies that a field evolution Φ of function type and domain E is built of functions $\texttt{snd}(\Phi(\varepsilon))$ which can take arguments of arbitrary domain, *including domains* $E' \nsubseteq E$. Notice that this property grants that a local denotational function value can be meaningfully moved around devices (through operators `nbr`, `rep`).

Notice that the definition of $\mathscr{V}[\![(T_1, \ldots, T_n) \to T]\!]$ by means of a function on whole field evolutions instead of local denotational values is required by the nature of the basic blocks of the language (`nbr`, `rep`), which cannot be computed pointwise event by event. We also remark that the denotation of a function type consists of *total* functions: this reflects the assumption that every function call is guaranteed to terminate (see Remark 3).

In the remainder of this paper, we use $\lambda \texttt{x} \in D.\texttt{f}$ to denote the mathematical function with domain D assigning each $\texttt{x} \in D$ to the corresponding value of expression $\texttt{f}$. We use $\Phi|_E$ for the restriction of the field evolution Φ to E, defined by $\lambda \varepsilon \in E.\Phi(\varepsilon)$ for denotational values Φ of local type and by $\lambda \varepsilon \in E.\Phi(\varepsilon)|_{E^{-}(\varepsilon)}$ for denotational values of field type. Whenever a sequence of field evolutions $\overline{\Phi}$ is assumed to share a common domain, we use $\mathbf{dom}(\overline{\Phi})$ with abuse of notation to denote their common domain.

Notice that we have not given an interpretation for parametric types containing free type variables $\bar{t}, \bar{l}, \bar{r}, \bar{s}$, even though these types are contemplated in the present system. Since the denotational semantics of parametric types is a well-understood topic (see e.g. [38]) and it is entirely orthogonal to the core semantics of field computations, we prefer for sake of clarity to give the definitions only for monomorphic types—extending those definition to polymorphic types would lead to a much heavier, though straightforward extension.[5]

4.3 Denotational semantics of expressions

The denotational semantics of a well-typed expression $\texttt{e}$ of type T in domain E under assumptions $X = \overline{\texttt{x}} \mapsto \overline{\Phi}$ is written $\mathscr{E}[\![\texttt{e}]\!]_X^E$ and yields a field evolution in $\mathscr{T}[\![T]\!]$ with domain E. As for the denotation of types, we assume that the denotations of built-in functions and constructors are given. In particular, this is represented by the function $\mathscr{C}[\![\texttt{c}]\!]$ in $(\mathscr{V}[\![L_1]\!] \times \ldots \times \mathscr{V}[\![L_n]\!]) \to \mathscr{V}[\![L]\!]$ translating the behaviour of built-in constructors $\texttt{c}$ of type $(\overline{L}) \to L$;[6] and by the function $\mathscr{B}[\![\texttt{b}]\!]$ in $\mathscr{T}[\![(\overline{T}) \to T]\!]$ translating the behaviour of built-in operators $\texttt{b}$ of type $(\overline{T}) \to T$ to denotational values (and possibly implicitly depending on sensor values and global environment status).

[5] In [38] this is achieved by setting $\mathscr{T}[\![\forall \bar{t}\bar{l}\bar{r}\bar{s}.T]\!] = \prod_{\bar{t} \in \mathbf{T}} \prod_{\bar{l} \in \mathbf{L}} \prod_{\bar{r} \in \mathbf{R}} \prod_{\bar{l} \in \mathbf{S}} \mathscr{T}[\![T]\!]$ where $\mathbf{T}$, $\mathbf{L}$, $\mathbf{R}$, $\mathbf{S}$ are the involved set of types. In other words, the elements of $\mathscr{T}[\![\forall \bar{t}\bar{l}\bar{r}\bar{s}.T]\!]$ are functions mapping all possible concrete instantiations of the type parameters to the denotational function values of the corresponding type.

[6] Since a constructor does not depend on the environment, we do not need an element of $\mathscr{V}[\![(\overline{L}) \to L]\!]$ in this case.

The interpretation function $\mathscr{E}[\![\cdot]\!]$ is then defined by the following rules:

$$
\begin{array}{rcl}
\mathscr{E}[\![\mathtt{x}]\!]^E_X & = & X(\mathtt{x})|_E \\
\mathscr{E}[\![\overline{\delta} \mapsto \overline{\mathtt{v}}]\!]^E_X & = & \lambda\varepsilon \in E.\overline{\delta} \mapsto \mathscr{E}[\![\overline{\mathtt{v}}]\!]^E_X(\varepsilon) \\
\mathscr{E}[\![\mathtt{c}(\overline{\ell})]\!]^E_X & = & \lambda\varepsilon \in E.\mathscr{C}[\![\mathtt{c}]\!](\mathscr{E}[\![\overline{\ell}]\!]^E_X(\varepsilon)) \\
\mathscr{E}[\![\mathtt{b}]\!]^E_X & = & \lambda\varepsilon \in E.\langle \mathtt{b},\ \mathscr{B}[\![\mathtt{b}]\!]\rangle \\
\mathscr{E}[\![\mathtt{d}]\!]^E_X & = & \lambda\varepsilon \in E.\langle \mathtt{d},\ \lim_n \mathscr{D}[\![\mathtt{d}]\!]_n\rangle \\
\mathscr{E}[\![(\overline{\mathtt{x}})\ \mathtt{=>}\ \mathtt{e}]\!]^E_X & = & \lambda\varepsilon \in E.\langle (\overline{\mathtt{x}})\ \mathtt{=>}\ \mathtt{e},\ \lambda\overline{\Phi} \in \mathscr{T}[\![\overline{T}]\!].\mathscr{E}[\![\mathtt{e}]\!]^{\mathbf{dom}(\overline{\Phi})}_{X\cup\overline{\mathtt{x}}\mapsto\overline{\Phi}}\rangle \\
\mathscr{E}[\![\mathtt{e}'(\overline{\mathtt{e}})]\!]^E_X & = & \lambda\varepsilon \in E.\mathtt{snd}\left(\mathscr{E}[\![\mathtt{e}']\!]^E_X(\varepsilon)\right)\left(\mathscr{E}[\![\overline{\mathtt{e}}]\!]^E_X|_{E(\mathtt{e}',\varepsilon)}\right)(\varepsilon) \\
\mathscr{E}[\![\mathtt{nbr}\{\mathtt{e}\}]\!]^E_X & = & \lambda\varepsilon \in E.\lambda\delta \in E^{-}(\varepsilon).\mathscr{E}[\![\mathtt{e}]\!]^E_X(\varepsilon^\delta) \\
\mathscr{E}[\![\mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}_2\}]\!]^E_X & = & \lim_n \mathscr{R}[\![\mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}_2\}]\!]_n
\end{array}
$$

Where:

- $\mathscr{D}[\![\mathtt{d}]\!]_n$ is the partial function translating the behaviour of $\mathtt{d}$ when recursion is bounded to depth n, and is defined by rules:

$$
\begin{array}{rcl}
\mathscr{D}[\![\mathtt{d}]\!]_0 & = & \emptyset \\
\mathscr{D}[\![\mathtt{d}]\!]_{n+1} & = & \lambda\overline{\Phi} \in \mathscr{T}[\![\overline{T}]\!].\mathscr{E}[\![\mathit{body}(\mathtt{d})]\!]^{\mathbf{dom}(\overline{\Phi})}_{X\cup \mathit{args}(\mathtt{d})\mapsto\overline{\Phi},\mathtt{d}\mapsto\lambda\varepsilon\in E.\mathscr{D}[\![\mathtt{d}]\!]_n}
\end{array}
$$

- $E(\mathtt{e}',\varepsilon)$ is equal to E if $\mathscr{E}[\![\mathtt{e}']\!]^E_X(\varepsilon)$ is a built-in operator, and to $\{\varepsilon' \in E :\ \mathscr{E}[\![\mathtt{e}']\!]^E_X(\varepsilon') = \mathscr{E}[\![\mathtt{e}']\!]^E_X(\varepsilon)\}$ otherwise (that is, the set of events in E aligned with ε with respect to the computation of $\mathtt{e}'$).

- $\mathscr{R}[\![\mathtt{e}]\!]_n$ with $\mathtt{e} = \mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}_2\}$ denotes the $\mathtt{rep}$ construct as bounded to n loop steps, and it is defined by rules:

$$
\begin{array}{rcl}
\mathscr{R}[\![\mathtt{e}]\!]_0 & = & \mathscr{E}[\![\mathtt{e}_1]\!]^E_X \\
\mathscr{R}[\![\mathtt{e}]\!]_{n+1} & = & \mathscr{E}[\![\mathtt{e}_2]\!]^E_{X\cup\mathtt{x}\mapsto\mathbf{shift}(\mathscr{R}[\![\mathtt{e}]\!]_n,\mathscr{R}[\![\mathtt{e}]\!]_0)}
\end{array}
$$

  where $\mathbf{shift}(\Phi,\Phi_0) = \lambda\varepsilon \in E.\Phi(\varepsilon^-)\mathtt{if}\ \varepsilon^-$ exists $\mathtt{else}\ \Phi_0(\varepsilon)$ pushes each value in Φ to the next future event, while falling back to Φ_0 for starting events. In the remainder of this paper, for ease of notation we shall drop the reference to Φ_0 and keep it implicit in the definition of $\mathbf{shift}(\cdot)$.

The rules above provide a definition of $\mathscr{E}[\![\cdot]\!]$ by induction on the structure of the expressions. In the remainder of this paper we shall feel free to omit the subscript X whenever $X = \emptyset$ and the superscript E whenever $E = \mathbf{E}$. Notice that syntactic values are always denoted by constant field evolutions, and can be reconstructed from their denotation (with possibly the exception of constructors).

The denotation of variables is straightforward, while the denotation of constructors and built-in operators is abstracted away assuming that corresponding $\mathscr{C}[\![\mathtt{c}]\!]$ and $\mathscr{B}[\![\mathtt{b}]\!]$ are given. In order to produce neighbouring field values with the correct domain, we require that in case the return type T of $\mathtt{b}$ is a *field* type, then $\mathbf{dom}(\mathscr{B}[\![\mathtt{b}]\!](\overline{\Phi})(\varepsilon)) = E^{-}(\varepsilon)$ for all possible $\overline{\Phi}$ in $\mathscr{T}[\![\overline{T}]\!]$ of domain E and $\varepsilon \in E$. Even though most built-in operators (pure operators, local sensors) could be defined pointwise in the same way constructors are defined, this is not possible for relational sensors (as $\mathtt{nbr\text{-}range}$) thus we opted for a more general and simpler formulation. The denotation of neighbouring field values is given for convenience, but since neighbouring field values does not occur in source programs is not needed for their denotation.

The denotation of defined functions, as usual, is defined as a *fixpoint* of an iterated process starting from the function $\mathscr{D}[\![\mathtt{d}]\!]_0$ with empty domain. At each subsequent step $n+1$, the body of $\mathtt{d}$ is evaluated with respect to the context that associates the name $\mathtt{d}$ itself with the previously obtained function $\mathscr{D}[\![\mathtt{d}]\!]_n$ (and the arguments of $\mathtt{d}$ with the respective values). We assume that the resulting function $\mathscr{D}[\![\mathtt{d}]\!]_{n+1}$ is undefined if it calls $\mathscr{D}[\![\mathtt{d}]\!]_n$ with arguments outside of its domain. It follows by easy induction that each such step is a conservative extension, i.e., $\mathscr{D}[\![\mathtt{d}]\!]_n \subseteq \mathscr{D}[\![\mathtt{d}]\!]_{n+1}$, hence the limit of the process is well-defined. Since function calls are guaranteed to terminate, this limit will be a total function as required by the denotation of function types (see Remark 3).

The denotation of a function application $\mathtt{e}'(\overline{\mathtt{e}})$ is given pointwise by event, and applies the second coordinate of $\mathscr{E}[\![\mathtt{e}']\!]^E_X(\varepsilon)$ (that is, the mathematical function corresponding to $\mathtt{e}'$) interpreted in the restricted domain $E(\mathtt{e}',\varepsilon)$ (computed through the first coordinate of $\mathscr{E}[\![\mathtt{e}']\!]^E_X$) containing ε to the arguments $\mathscr{E}[\![\overline{\mathtt{e}}]\!]^E_X$. Such domain restriction is used in order to prevent interference among non-aligned devices: in fact, no restriction is needed for built-in operators while restriction to devices computing the same function $\mathtt{e}'$ is needed otherwise. The importance of this aspect shall be further clarified in the following sections. As the rule above is formulated, it seems that a whole field evolution Φ is calculated for each event ε, while being used only to produce the local value $\Phi(\varepsilon)$. However, the whole field evolution is actually used since each event in its domain $E(\mathtt{e}',\varepsilon)$ computes the same function on the same arguments, hence producing the same output field evolution. Thus the rule could also be reformulated as follows:

$$\mathscr{E}[\![\mathtt{e}'(\overline{\mathtt{e}})]\!]^E_X = \bigcup_{\varepsilon\in E} \mathtt{snd}\left(\mathscr{E}[\![\mathtt{e}']\!]^E_X(\varepsilon)\right)\left(\mathscr{E}[\![\overline{\mathtt{e}}]\!]^E_X|_{E(\mathtt{e}',\varepsilon)}\right)$$

The denotation of operator `nbr` yields in each event ε a neighbouring field of domain $E^{-}(\varepsilon)$ mapping to the values of expression `e` in the corresponding events.

The denotation of operator `rep` is carried out by a fixpoint process as for recursive functions. First, a field evolution $\mathscr{R}[\![\cdot]\!]_0$ is computed holding the initial values computed by $\mathtt{e}_1$ in each event. At each subsequent step, the results computed by $\mathscr{R}[\![\cdot]\!]_n = \Phi$ in each event are made available to their subsequent events through the new assumption $\mathtt{x} \mapsto \mathbf{shift}(\Phi)$ in X. It follows that once the value at each event in $E^{-}(\varepsilon)$ stabilizes, the value at ε also stabilizes in one more iteration. Since the events form a DAG and values at source events (events without predecessor) are steadily equal to the initial value by construction, the whole process stabilizes after a number of iterations at most equal to the cardinality of $\mathbf{E}$, hence the limit of the process is well-defined.

4.4 Example

We now illustrate the denotational semantics by applying it to representative example expressions.

Distance-To

For a first example, consider the expression $\mathtt{e}_{dist}$, computing the distance of every device from devices in a given source set indicated by the Boolean-valued field `s`:

```
mux( s, 0, min-hood+(nbr-range() + nbr{d}) )   ;;  e'
rep (infinity) { (d) => e' }   ;;  e_dist
```

where `min-hood+` is a built-in function that returns the minimum value amongst a device's neighbours, excluding itself. Figure 7 shows the evaluation of the denotational semantics for this expression, as evaluated with respect to the neighbourhood graph shown in Figure 6. We consider as input a source set `s` consisting of device 2 before its reboot, and device 1 beginning at its fourth firing, represented by the Boolean field evolution $\Phi_{\mathtt{s}}$, with corresponding environment $X = \mathtt{s} \mapsto \Phi_{\mathtt{s}}$, shown in the top left of Figure 7 (b). We assume the devices to be moving[7] so that their relative distance changes over time as depicted in the center left of Figure 7 (a).

The outermost component of expression $\mathtt{e}_{dist}$ is a `rep` operator, thus $\mathscr{E}[\![\mathtt{e}_{dist}]\!]_X$ is calculated via the following procedure:

- First, $\mathscr{R}[\![\mathtt{e}_{dist}]\!]_0$ is calculated as $\mathscr{E}[\![\infty]\!]_X = \Phi_0$ (since ∞ is the initial value of the `rep`-expression), a constant field evolution.
- This value is then shifted in time (in this case leaving it unchanged, $\mathbf{shift}(\Phi_0) = \Phi_0$) and incorporated in the substitution $X_0 = X \cup \mathtt{d} \mapsto \Phi_0$. Thus $\mathscr{R}[\![\mathtt{e}_{dist}]\!]_1$ is calculated as $\mathscr{E}[\![\mathtt{e}']\!]_{X_0}$ giving a new field evolution Φ_1. This evaluation is illustrated step-by-step in Figure 7 (a) and (b) top and center, breaking $\mathtt{e}'$ into all its subexpressions.

[7] Note that the x-axis in Figure 7 is indexed by *device* and not by *position*.

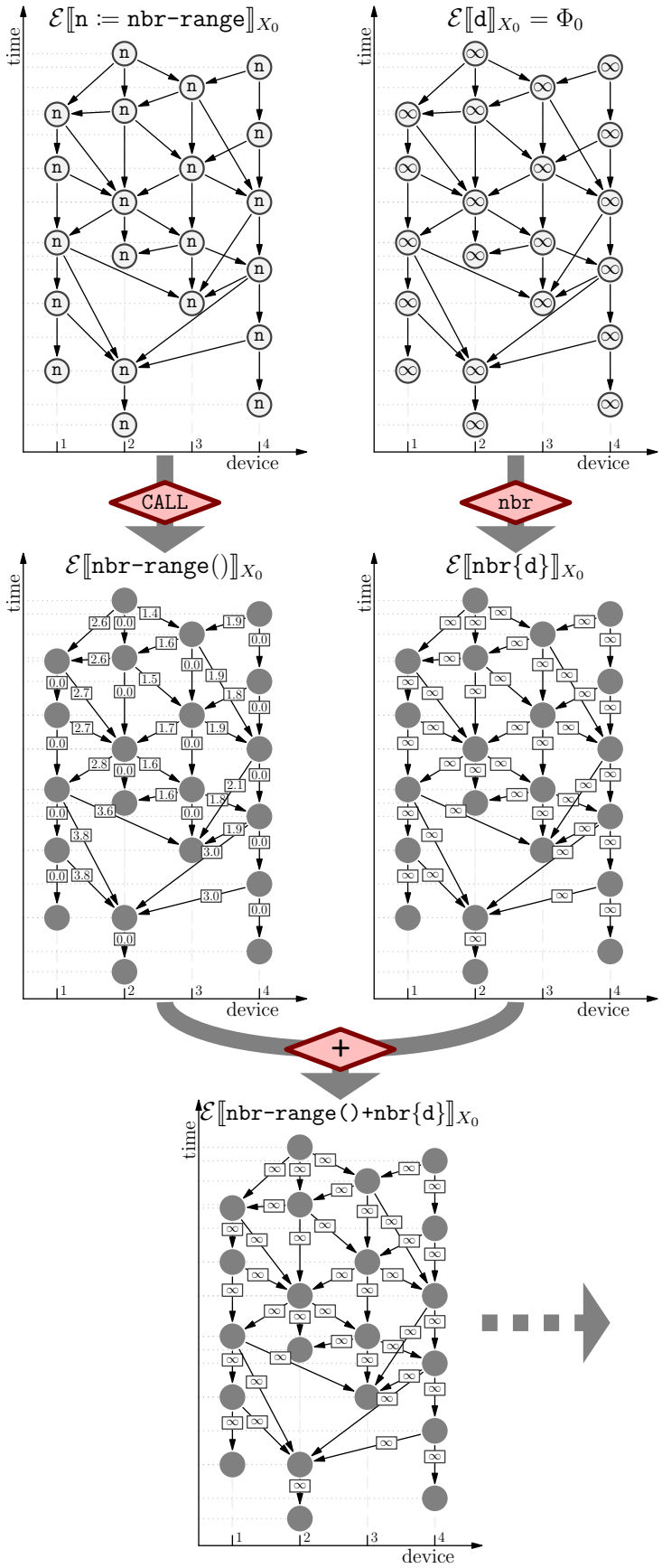


Figure 7: (a) The denotational semantics of a distance-to calculation.

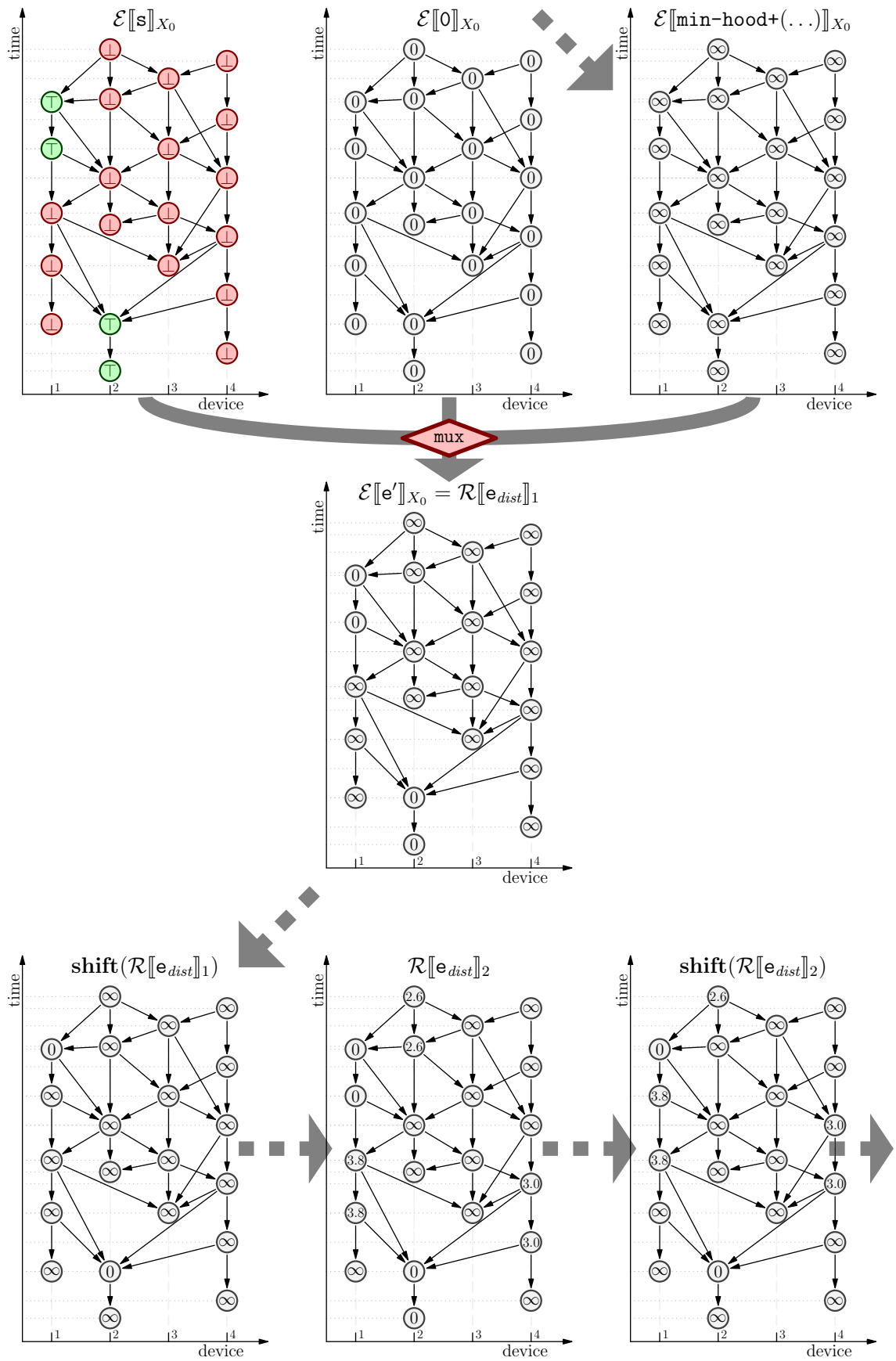


Figure 7: (b) The denotational semantics of a distance-to calculation (cont.)

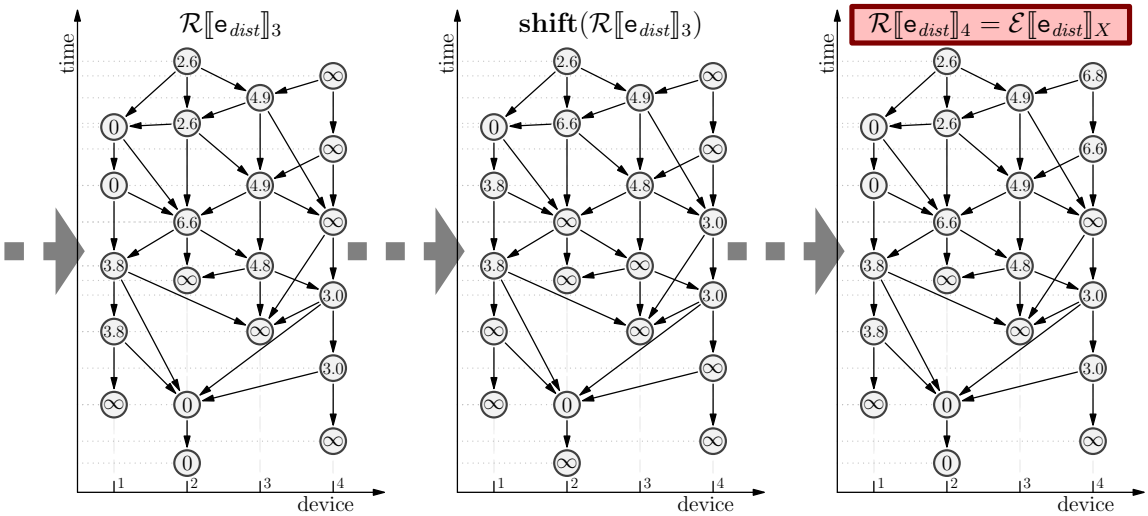


Figure 7: (c) The denotational semantics of a distance-to calculation (cont.)

- The process of shift and evaluation is then repeated: Φ_1 is shifted in time (Figure 7 (b) bottom left), incorporated in a new substitution $X_1 = X \cup \mathtt{d} \mapsto \mathbf{shift}(\Phi_1)$ and $\mathscr{E}[\![\mathsf{e}']\!]_{X_1}$ is expanded into another field evolution Φ_2 (Figure 7 (b) bottom center). Shifting and evaluation continues until a fixed point is reached, which in the case of this example happens at stage $n = 4$ (Figure 7 (c) right).

Notice that due to the characteristics of the `rep` operator, the values `nbr{d}` collected from neighbour devices are not the latest but instead the ones before them (that is, the values fed to the update function in the latest event). The latest outcome of a `rep` operator could instead be obtained via `nbr{rep(·){·}}`, but this construct is not used here as that would not allow the distance calculation to propagate across multiple hops in the network. This additional delay sometimes leads to counterintuitive behaviours: for example, the third firing ε of device 3 calculates gradient 4.9 obtained through device 4, which however holds value ∞ in its latest firing available to ε. In fact, the value to which ε refers to is the previous one, which is equal to 3.0. This behaviour can slow down the propagation of updates through a network, but appears necessary for ensuring both safe and general composition.

Distance avoiding obstacles

The previous example allowed us to show the denotation of data values (`nbr-range`, 0), variable lookups (`d`, `s`), builtin functions (`nbr-range`, `+[f,f]`, `mux`, `min-hood+`), `nbr` and `rep` constructs. It did not, however, show an example of *branching*, which in the present calculi is modeled by function calls $\mathtt{e}(\overline{\mathtt{e}})$ where the functional expression `e` is *not* constant.

To illustrate branching, we now expand on the previous example by considering the expression $\mathtt{e}_{avoid}$ which computes the distance of each device from a given source set *avoiding some obstacles*:

```
def f(s) { infinity }
def g(s) { e_dist }
e_avoid mux(avoid, f, g)(source)   ;; e_avoid
```

The denotational semantics of $\mathtt{e}_{avoid}$ on the sample network in Figure 6 is shown in Figure 8. We consider as input the same source set `source` (with corresponding field evolution $\Phi_{\mathtt{s}}$) as in the previous example, and a set of obstacles `avoid` corresponding to the first firings of device 3 and device 2 after its reboot (blue nodes in Figure 8 top left), together enclosed in environment X.

When the field of functions (top left) is called on the argument (top right), the computation branches in two parts:

- the events holding f, which just compute the constant value ∞ (center left);

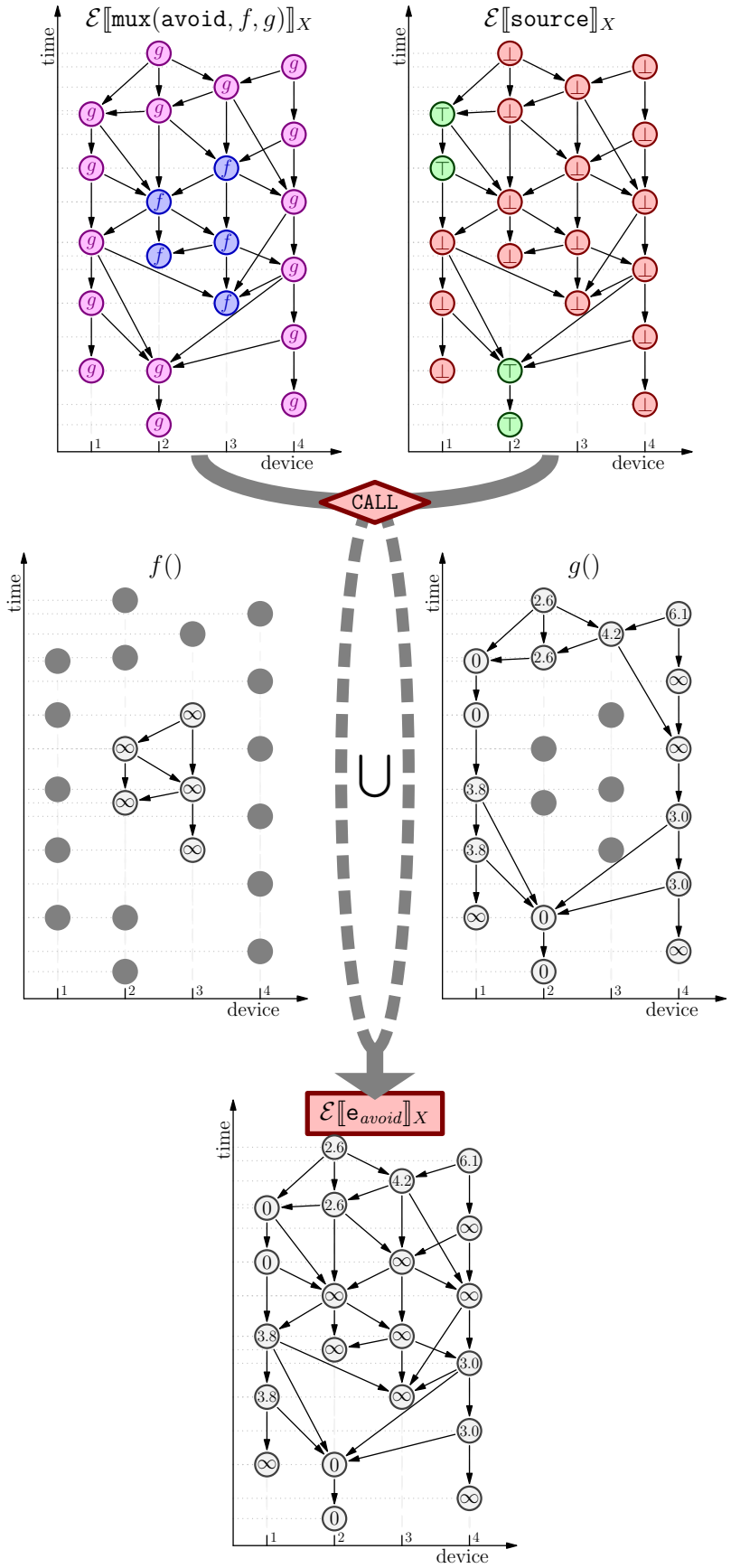


Figure 8: The denotational semantics of a gradient calculation avoiding obstacles.

- the events holding g, which compute a distance from the source set, as in the previous example (center right).

Notice that since the events holding g compute their distances in isolation, their final values differ from the ones obtained in the previous example.

Finally, the two different branches are merged together in order to form the final outcome of the function call, in Figure 8 bottom.

4.5 Properties

The denotational semantics can be used in order to formulate conveniently intuitive facts about the calculus.

Alignment

Given any expression $\mathtt{e}$ of field type, $\mathscr{E}[\![\mathtt{e}]\!]^E(\varepsilon)$ is a neighbouring field with domain $E^{-}(\varepsilon)$. This fact can be easily checked in the rule for `nbr`. In the case of function application, notice that $\mathtt{e}'$ has to be a value since it has type $(\overline{T}) \to F$. Thus $E(\mathtt{e}', \varepsilon) = E$ and the thesis follows by inductive hypothesis using `nbr` and built-in operators as base case.

Restriction

Given any expression $\mathtt{e}_0$ executed in domain E and $V = \mathscr{E}[\![\mathtt{e}_0]\!]^E(\varepsilon)$ for some $\varepsilon \in E$, we say that $(\mathscr{E}[\![\mathtt{e}_0]\!]^E)^{-1}(V) = E(\mathtt{e}_0, \varepsilon)$ is a cluster. We say that function call has the restriction property to mean that well-typed expression $\mathtt{e}_0(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ computes in isolation in each such cluster.

Namely, given $\mathtt{e}_0$ and any of its clusters $E(\mathtt{e}_0, \varepsilon)$, let $\mathtt{e}_1, \ldots, \mathtt{e}_n$ and $\mathtt{e}'_1, \ldots, \mathtt{e}'_n$ be such that the denotation of $\mathtt{e}_i$ coincides with that of $\mathtt{e}'_i$ on $E(\mathtt{e}_0, \varepsilon)$, that is $\mathscr{E}[\![\mathtt{e}_i]\!]^E|_{E(\mathtt{e}_0,\varepsilon)} = \mathscr{E}[\![\mathtt{e}'_i]\!]^E|_{E(\mathtt{e}_0,\varepsilon)}$. Then $\mathscr{E}[\![\mathtt{e}_0(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]^E(\varepsilon) = \mathscr{E}[\![\mathtt{e}_0(\mathtt{e}'_1, \ldots, \mathtt{e}'_n)]\!]^E(\varepsilon)$ for any $\varepsilon \in E(\mathtt{e}_0, \varepsilon)$.

This means that computation of $\mathtt{e}_0$ inside cluster $E(\mathtt{e}_0, \varepsilon)$ is independent of the outcome of computation outside it, and of events outside it. This fact reflects the intuition beyond function application given in Section 3, and implies the analogous property for the first-order calculus with conditionals (since the conditional expression can be simulated by function application).

Repeating statements

Consider now an expression $\mathtt{e} = \mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x}) \mathtt{=>} \mathtt{e}_2\}$ and suppose that ε is a "source" event in E, that is, there exists no ε' in E such that $\mathit{neigh}(\varepsilon, \varepsilon')$. Then $\mathscr{E}[\![\mathtt{e}]\!]^E(\varepsilon) = \mathscr{E}[\![((\mathtt{x}) \mathtt{=>} \mathtt{e}_2)(\mathtt{e}_1)]\!]^E$.

Furthermore, assume now that $\mathtt{e}$ does not contain `nbr` statements or non-pure operators and ε is any event in E. Then $\mathscr{E}[\![\mathtt{e}]\!]^E(\varepsilon) = \mathscr{E}[\![\mathtt{e}]\!]^{E'}(\varepsilon)$ where E' is the subset of E containing only the events on device δ_ε.

5 Operational Semantics

Having established a denotational semantics describing the behavior of the aggregate of devices in the previous section, we now develop an operational semantics that describes the equivalent computation as carried out by an individual device at a particular event, and hence, by a whole network of devices over time. In particular, this section presents a formal semantics that can serve as a specification for implementation of programming languages based on the calculus.

5.1 Device Semantics (Big-Step Operational Semantics)

According to the "local" viewpoint, individual devices undergo computation in rounds. In each round, a device sleeps for some time, wakes up, gathers information about messages received from neighbours while sleeping, performs an evaluation of the program, and finally emits a message to all neighbours with information about the outcome of computation before going back to sleep. The scheduling of such rounds across the network is fair and non-synchronous.

We base operational semantics on the syntax introduced in Section 3.1 (Figure 2), To simplify the notation, we shall assume a fixed program P. We say that "device δ *fires*", to mean that the main expression of P is evaluated on δ at a particular round.

We model device computation by a big-step operational semantics where the result of evaluation is a *value-tree* θ, which is an ordered tree of values that tracks the results of all evaluated subexpressions. 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 any implementation might massively compress the value-tree, storing only informations about sub-expressions which are relevant for the computation). A *value-tree environment* Θ is a map from device identifiers to value-trees, collecting the outcome of the last evaluation on the neighbours. This is written $\overline{\delta} \mapsto \overline{\theta}$ 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 9 (first frame). Figure 9 (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 *args* and *body* for extracting the formal parameters and the body of a (user-defined or anonymous) function, respectively. The computation that takes place on a single device is formalised by the big-step operational semantics rules given in Figure 9 (fourth frame). The derived judgements are of the form $\delta; \Theta \vdash \mathtt{e} \Downarrow \theta$, to be read "expression e evaluates to value-tree θ on device δ with respect to the value-tree environment Θ", where: *(i)* δ is the identifier of the current device; *(ii)* Θ is the neighbouring field of the value-trees produced by the most recent evaluation of (an expression corresponding to) e on δ's neighbours; *(iii)* e is a run-time expression (i.e., an expression that may contain neighbouring field values); *(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 resulting value of expression e.

The operational semantics rules are based on rather standard rules for functional languages, extended so as to be able to evaluate a subexpression $\mathtt{e}'$ 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 modeled by the auxiliary function π, defined in Figure 9 (third frame). The 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 $\pi^{\ell}(\theta)$ extracts the last subtree of θ, if it is present and the root of the second last subtree of θ is equal to the local value ℓ.

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 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 restriction is obeyed (cf. Section 2), 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 $\mathtt{e}_{n+1}(\mathtt{e}_1 \cdots \mathtt{e}_n)$ such that the evaluation of $\mathtt{e}_{n+1}$ produces a value-tree θ_{n+1} whose root $\rho(\theta_{n+1})$ is a built-in function b. It produces the value-tree $\mathtt{v}\langle\theta_1, \ldots, \theta_n, \theta_{n+1}\rangle$, where $\theta_1, \ldots, \theta_{n+1}$ are the value-trees produced by the evaluation of the actual parameters and functional expression $\mathtt{e}_1, \ldots, \mathtt{e}_{n+1}$ ($n \geq 0$) and v is the value returned by the function. Rule [E-B-APP] exploits the special auxiliary function $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}$, whose actual definition is abstracted away. This is such that $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}(\overline{\mathtt{v}})$ computes the result of applying built-in function b to values $\overline{\mathtt{v}}$ in the current environment of the device δ. As in the denotational case, we require that $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}$ always yields values of the expected type T where $\mathtt{b} : (\overline{T}) \to T \in \mathscr{B}$.

In particular, for the examples in this paper, we assume that the built-in 0-ary function uid gets evaluated to the current device identifier (i.e., $(\!|\mathtt{uid}|\!)^{\Theta}_{\delta}() = \delta$), and that mathematical operators have their standard meaning, which is independent from δ and Θ (e.g., $(\!|+|\!)^{\Theta}_{\delta}(1,2) = 3$). We also assume that map-hood, fold-hood reflect the rules for function application, so that for instance $\mathtt{map\text{-}hood}(\mathtt{f}, \overline{\delta} \mapsto \overline{\mathtt{v}}) = \overline{\delta} \mapsto \mathtt{f}(\overline{\mathtt{v}})$ (where $\mathtt{f}(\mathtt{v}_i)$ is computed w.r.t. the empty value-tree environment $\Theta = \emptyset$). The $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}$ function also encapsulates measurement variables such as nbr-range and interactions with the external world via sensors and actuators.

In order to ensure that domain restriction is obeyed, for each built-in function b we assume that $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}(\mathtt{v}_1, \cdots, \mathtt{v}_n)$ is defined only if all the neighbouring field values in $\mathtt{v}_1, \ldots, \mathtt{v}_n$ have domain $\mathbf{dom}(\Theta) \cup \{\delta\}$; and if $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}(\mathtt{v}_1, \cdots, \mathtt{v}_n)$ returns a neighbouring field value ϕ, then $\mathbf{dom}(\phi) = \mathbf{dom}(\Theta) \cup \{\delta\}$. For example, evaluating the expression

Value-trees and value-tree environments:

$\theta \quad ::= \quad \mathtt{v}\langle\overline{\theta}\rangle$ — value-tree

$\Theta \quad ::= \quad \overline{\delta} \mapsto \overline{\theta}$ — value-tree environment

Auxiliary functions:

$\rho(\mathtt{v}\langle\overline{\theta}\rangle) = \mathtt{v}$

$\pi_i(\mathtt{v}\langle\theta_1,\ldots,\theta_n\rangle) = \theta_i \quad \text{if } 1 \le i \le n$ $\qquad$ $\pi^\ell(\mathtt{v}\langle\theta_1,\ldots,\theta_{n+2}\rangle) = \theta_{n+2} \quad \text{if } \rho(\theta_{n+1}) = \ell$

$\pi_i(\theta) = \bullet \quad \text{otherwise}$ $\qquad$ $\pi^\ell(\theta) = \bullet \quad \text{otherwise}$

$$\text{For } aux \in \rho, \pi_i, \pi^\ell : \quad \begin{cases} aux(\delta \mapsto \theta) & = & \delta \mapsto aux(\theta) & \text{if } aux(\theta) \neq \bullet \\ aux(\delta \mapsto \theta) & = & \bullet & \text{if } aux(\theta) = \bullet \\ aux(\Theta, \Theta') & = & aux(\Theta), aux(\Theta') & \end{cases}$$

$args(\mathtt{d}) = \overline{\mathtt{x}} \quad \text{if } \mathtt{def\ d}(\overline{\mathtt{x}})\ \{\mathtt{e}\}$ $\qquad$ $body(\mathtt{d}) = \mathtt{e} \quad \text{if } \mathtt{def\ d}(\overline{\mathtt{x}})\ \{\mathtt{e}\}$

$args((\overline{\mathtt{x}})\ \mathtt{=>}\ \mathtt{e}) = \overline{\mathtt{x}}$ $\qquad$ $body((\overline{\mathtt{x}})\ \mathtt{=>}\ \mathtt{e}) = \mathtt{e}$

Syntactic shorthands:

$\delta;\overline{\pi}(\Theta) \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta}$	where $\lvert\overline{\mathtt{e}}\rvert = n$	for $\delta;\pi_1(\Theta) \vdash \mathtt{e}_1 \Downarrow \theta_1 \quad \cdots \quad \delta;\pi_n(\Theta) \vdash \mathtt{e}_n \Downarrow \theta_n$
$\rho(\overline{\theta})$	where $\lvert\overline{\theta}\rvert = n$	for $\rho(\theta_1),\ldots,\rho(\theta_n)$
$\overline{\mathtt{x}} := \rho(\overline{\theta})$	where $\lvert\overline{\mathtt{x}}\rvert = n$	for $\mathtt{x}_1 := \rho(\theta_1)\ \ldots\ \mathtt{x}_n := \rho(\theta_n)$

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

$$\text{[E-LOC]}\ \frac{}{\delta;\Theta \vdash \ell \Downarrow \ell\langle\rangle} \qquad \text{[E-FLD]}\ \frac{\phi' = \phi|_{\mathbf{dom}(\Theta)\cup\{\delta\}}}{\delta;\Theta \vdash \phi \Downarrow \phi'\langle\rangle}$$

$$\text{[E-B-APP]}\ \frac{\delta;\overline{\pi}(\Theta) \vdash \overline{\mathtt{e}}, \mathtt{e} \Downarrow \overline{\theta}, \theta \qquad \mathtt{v} = (\!|\rho(\theta)|\!)^{\Theta}_{\delta}(\rho(\overline{\theta}))}{\delta;\Theta \vdash \mathtt{e}(\overline{\mathtt{e}}) \Downarrow \mathtt{v}\langle\overline{\theta}, \theta\rangle}$$

$$\text{[E-D-APP]}\ \frac{\delta;\overline{\pi}(\Theta) \vdash \overline{\mathtt{e}}, \mathtt{e} \Downarrow \overline{\theta}, \theta \quad \mathtt{f} = \rho(\theta) \quad \delta;\pi^{\mathtt{f}}(\Theta) \vdash body(\mathtt{f})[args(\mathtt{f}) := \rho(\overline{\theta})] \Downarrow \theta'}{\delta;\Theta \vdash \mathtt{e}(\overline{\mathtt{e}}) \Downarrow \rho(\theta')\langle\overline{\theta}, \theta, \theta'\rangle}$$

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

$$\text{[E-REP]}\ \frac{\begin{array}{lll} \delta;\pi_1(\Theta) \vdash \mathtt{e}_1 \Downarrow \theta_1 & \ell_1 = \rho(\theta_1) & \\ \delta;\pi_2(\Theta) \vdash \mathtt{e}_2[\mathtt{x} := \ell_0] \Downarrow \theta_2 & \ell_2 = \rho(\theta_2) & \end{array} \quad \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathbf{dom}(\Theta) \\ \ell_1 & \text{otherwise} \end{cases}}{\delta;\Theta \vdash \mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}_2\} \Downarrow \ell_2\langle\theta_1, \theta_2\rangle}$$

Figure 9: Big-step operational semantics for expression evaluation.

$+(1\ 2)$ produces the value-tree $3\langle 1\langle\rangle, 2\langle\rangle, +\langle\rangle\rangle$. The value of the whole expression, 3, has been computed by using rule [E-B-APP] to evaluate the application of the sum operator $+$ (the root of the third subtree of the value-tree) to the values 1 (the root of the first subtree of the value-tree) and 2 (the root of the second subtree of the value-tree). In the following, for sake of readability, we sometimes write the value $\mathtt{v}$ as short for the value-tree $\mathtt{v}\langle\rangle$. Following this convention, the value-tree $3\langle 1\langle\rangle, 2\langle\rangle, +\langle\rangle\rangle$ is shortened to $3\langle 1, 2, +\rangle$.

Rule [E-D-APP] models the application of user-defined or anonymous functions, i.e., it is used to evaluate expressions of the form $\mathtt{e}_{n+1}(\mathtt{e}_1 \cdots \mathtt{e}_n)$ such that the evaluation of $\mathtt{e}_{n+1}$ produces a value-tree θ_{n+1} whose root $\ell = \rho(\theta_{n+1})$ is a user-defined function name or an anonymous function. It is similar to rule [E-B-APP], however it produces a value-tree which has one more subtree, θ_{n+2}, which is produced by evaluating the body of the function ℓ with respect to the value-tree environment $\pi^\ell(\Theta)$ containing only the value-trees associated to the evaluation of the body of the same function ℓ.

To illustrate rule [E-REP] (`rep` construct), as well as computational rounds, we consider program `rep(0){(x) => +(x, 1)}` (cf. Section 3.1). The first firing of a device δ after activation or reset is performed against the empty tree environment. Therefore, according to rule [E-REP], to evaluate `rep(0){(x) => +(x, 1)}` means to evaluate the subexpression `+(0, 1)`, obtained from `+(x, 1)` by replacing `x` with `0`. This produces the value-tree

$\theta = 1\langle 0, 1\langle 0, 1, +\rangle\rangle$, where root 1 is the overall result as usual, while its sub-trees are the result of evaluating the first and second argument respectively. Any subsequent firing of the device δ is performed with respect to a tree environment Θ that associates to δ the outcome of the most recent firing of δ. Therefore, evaluating `rep(0){(x) => +(x, 1)}` at the second firing means to evaluate the subexpression `+(1, 1)`, obtained from `+(x, 1)` by replacing `x` with `1`, which is the root of θ. Hence the results of computation are 1, 2, 3, and so on.

Notice that in both rules [E-REP], [E-NBR] we do not assume that Θ is empty whenever it does not contain δ. This might seem unnatural at a first glance, since every time a device is restarted its first firing is computed with respect to the empty value-tree environment, and all the subsequent firings will contain δ their domains. However, this fact is not inductively true for the sub-expressions of $\mathtt{e}_{\mathtt{main}}$: for example, the first time a conditional guard evaluates to `True` the if-expression will be evaluated w.r.t. an environment not containing δ but possibly containing other devices whose guard evaluated to `True` in their previous round of computation.

Value-trees also support modelling information exchange through the `nbr` construct, as of rule [E-NBR]. Consider the program $\mathtt{e}' = \mathtt{min\text{-}hood}(\mathtt{nbr}\{\mathtt{sns\text{-}num}()\})$, where the 1-ary built-in function `min-hood` returns the lower limit of values in the range of its neighbouring field argument, and the 0-ary built-in function `sns-num` returns the numeric value measured by a sensor. Suppose that the program runs on a network of three fully connected devices δ_A, δ_B, and δ_C where `sns-num` returns `1` on δ_A, `2` on δ_B, and `3` on δ_C. Considering an initial empty tree-environment $\emptyset$ on all devices, we have the following: the evaluation of $\mathtt{sns\text{-}num}()$ on δ_A yields $1\langle\mathtt{sns\text{-}num}\rangle$ (by rules [E-LOC] and [E-B-APP], since $(\!|\mathtt{sns\text{-}num}|\!)^{\emptyset}_{\delta_A}() = 1$); the evaluation of $\mathtt{nbr}\{\mathtt{sns\text{-}num}()\}$ on δ_A yields $(\delta_A \mapsto 1)\langle 1\langle\mathtt{sns\text{-}num}\rangle\rangle$ (by rule [E-NBR]); and the evaluation of $\mathtt{e}'$ on δ_A yields

$$\theta_A \;=\; 1\langle(\delta_A \mapsto 1)\langle 1\langle\mathtt{sns\text{-}num}\rangle\rangle, \mathtt{min\text{-}hood}\rangle$$

(by rule [E-B-APP], since $(\!|\mathtt{min\text{-}hood}|\!)^{\emptyset}_{\delta_A}(\delta_A \mapsto 1) = 1$). Therefore, after its first firing, device δ_A produces the value-tree θ_A. Similarly, after their first firing, devices δ_B and δ_C produce the value-trees

$$\begin{array}{rcl}\theta_B & = & 2\langle(\delta_B \mapsto 2)\langle 2\langle\mathtt{sns\text{-}num}\rangle\rangle, \mathtt{min\text{-}hood}\rangle \\ \theta_C & = & 3\langle(\delta_C \mapsto 3)\langle 3\langle\mathtt{sns\text{-}num}\rangle\rangle, \mathtt{min\text{-}hood}\rangle\end{array}$$

respectively. Suppose that device δ_B is the first device that fires a second time. Then the evaluation of $\mathtt{e}'$ on δ_B is now performed with respect to the value tree environment $\Theta_B = (\delta_A \mapsto \theta_A,\ \delta_B \mapsto \theta_B,\ \delta_C \mapsto \theta_C)$ and the evaluation of its subexpressions $\mathtt{nbr}\{\mathtt{sns\text{-}num}()\}$ and $\mathtt{sns\text{-}num}()$ is performed, respectively, with respect to the following value-tree environments obtained from Θ_B by alignment:

$$\begin{array}{rcl}\Theta'_B & = & \pi_1(\Theta_B) \;=\; (\delta_A \mapsto (\delta_A \mapsto 1)\langle 1\langle\mathtt{sns\text{-}num}\rangle\rangle,\ \delta_B \mapsto \cdots,\ \delta_C \mapsto \cdots) \\ \Theta''_B & = & \pi_1(\Theta'_B) \;=\; (\delta_A \mapsto 1\langle\mathtt{sns\text{-}num}\rangle,\ \delta_B \mapsto 2\langle\mathtt{sns\text{-}num}\rangle,\ \delta_C \mapsto 3\langle\mathtt{sns\text{-}num}\rangle)\end{array}$$

We have that $(\!|\mathtt{sns\text{-}num}|\!)^{\Theta''_B}_{\delta_B}() = 2$; the evaluation of $\mathtt{nbr}\{\mathtt{sns\text{-}num}()\}$ on δ_B with respect to Θ'_B yields $\phi\langle 2\langle\mathtt{sns\text{-}num}\rangle\rangle$ where $\phi = (\delta_A \mapsto 1, \delta_B \mapsto 2, \delta_C \mapsto 3)$; and $(\!|\mathtt{min\text{-}hood}|\!)^{\Theta_B}_{\delta_B}(\phi) = 1$. Therefore the evaluation of $\mathtt{e}'$ on δ_B produces the value-tree $1\langle\phi\langle 2\langle\mathtt{sns\text{-}num}\rangle\rangle, \mathtt{min\text{-}hood}\rangle$. Namely, the computation at device δ_B after the first round yields 1, which is the minimum of `sns-num` across neighbours—and similarly for δ_A and δ_C.

We now present an example illustrating first-class functions. Consider the program $\mathtt{pick\text{-}hood}(\mathtt{nbr}\{\mathtt{sns\text{-}fun}()\})$, where the 1-ary built-in function `pick-hood` returns at random a value in the range of its neighbouring field argument, and the 0-ary built-in function `sns-fun` returns a 0-ary function returning a value of type `num`. Suppose that the program runs again on a network of three fully connected devices δ_A, δ_B, and δ_C where `sns-fun` returns $\ell_0 = ()\mathtt{=>}0$ on δ_A and δ_B, and returns $\ell_1 = ()\mathtt{=>}\mathtt{e}'$ on δ_C, where $\mathtt{e}' = \mathtt{min\text{-}hood}(\mathtt{nbr}\{\mathtt{sns\text{-}num}()\})$ is the program illustrated in the previous example. Assume that `sns-num` returns `1` on δ_A, `2` on δ_B, and `3` on δ_C. Then after its first firing, device δ_A produces the value-tree

$$\theta'_A \;=\; 0\langle\ell_0\langle(\delta_A \mapsto \ell_0)\langle\ell_0\langle\mathtt{sns\text{-}fun}\rangle\rangle, \mathtt{pick\text{-}hood}\rangle, 0\rangle$$

where the root of the first subtree of θ'_A is the anonymous function ℓ_0 (defined above), and the second subtree of θ'_A, 0, has been produced by the evaluation of the body 0 of ℓ_0. After their first firing, devices δ_B and δ_C produce the value-trees

$$\begin{array}{rcl}\theta'_B & = & 0\langle\ell_0\langle(\delta_B \mapsto \ell_0)\langle\ell_0\langle\mathtt{sns\text{-}fun}\rangle\rangle, \mathtt{pick\text{-}hood}\rangle, 0\rangle \\ \theta'_C & = & 3\langle\ell_1\langle(\delta_C \mapsto \ell_1)\langle\ell_1\langle\mathtt{sns\text{-}fun}\rangle\rangle, \mathtt{pick\text{-}hood}\rangle, \theta_C\rangle\end{array}$$

System configurations and action labels:

Ψ	$::=$	$\overline{\delta} \mapsto \overline{\Theta}$	status field
τ	$::=$	$\overline{\delta} \mapsto \overline{I}$	topology
Σ	$::=$	$\overline{\delta} \mapsto \overline{\sigma}$	sensors-map
Env	$::=$	τ, Σ	environment
N	$::=$	$\langle Env; \Psi \rangle$	network configuration
act	$::=$	$\delta \mid env$	action label

Environment well-formedness:

$WFE(\tau, \Sigma)$ holds if τ, Σ have same domain, and τ's values do not escape it.

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

$$\text{[N-FIR]} \quad \frac{Env = \tau, \Sigma \quad \tau(\delta) = \overline{\delta} \quad \delta; F(\Psi)(\delta) \vdash \mathtt{e}_{\mathtt{main}} \Downarrow \theta \text{ (w.r.t. } \Sigma(\delta)) \quad \Psi_1 = \overline{\delta} \mapsto \{\delta \mapsto \theta\}}{\langle Env; \Psi \rangle \xrightarrow{\delta} \langle Env; F(\Psi)[\Psi_1] \rangle}$$

$$\text{[N-ENV]} \quad \frac{WFE(Env') \quad Env' = \tau, \overline{\delta} \mapsto \overline{\sigma} \quad \Psi_0 = \overline{\delta} \mapsto \emptyset}{\langle Env; \Psi \rangle \xrightarrow{env} \langle Env'; \Psi_0[\Psi] \rangle}$$

Figure 10: Small-step operational semantics for network evolution.

respectively, where θ_C is the value-tree for $\mathtt{e}$ given in the previous example.

Suppose that device δ_A is the first device that fires a second time, and its `pick-hood` selects the function shared by device δ_C. The computation is performed with respect to the value tree environment $\Theta'_A = (\delta_A \mapsto \theta'_A,\ \delta_B \mapsto \theta'_B,\ \delta_C \mapsto \theta'_C)$ and produces the value-tree $1\langle \ell_1 \langle \phi' \langle \ell_1 \langle \mathtt{sns\text{-}fun} \rangle \rangle, \mathtt{pick\text{-}hood} \rangle, \theta''_A \rangle$, where $\phi' = (\delta_A \mapsto \ell_1, \delta_C \mapsto \ell_1)$ and $\theta''_A = 1\langle (\delta_A \mapsto 1, \delta_C \mapsto 3) \langle 1 \langle \mathtt{sns\text{-}num} \rangle \rangle, \mathtt{min\text{-}hood} \rangle$, since, according to rule [E-D-APP], the evaluation of the body $\mathtt{e}'$ of ℓ_1 (which produces the value-tree θ''_A) is performed with respect to the value-tree environment $\pi^{\ell_1}(\Theta'_A) = (\delta_C \mapsto \theta_C)$. Namely, device δ_A executed the anonymous function ℓ_1 received from δ_C, and this was able to correctly align with execution of ℓ_1 at δ_C, gathering values perceived by `sns-num` of 1 at δ_A and 3 at δ_C.

5.2 Network Semantics (Small-Step Operational Semantics)

We now provide an operational semantics for the evolution of whole networks, namely, for modelling the distributed evolution of computational fields over time. Figure 10 (top) defines key syntactic elements to this end. Ψ models the overall status of the devices in the network at a given time, as a map from device identifiers to value-tree environments. From it we can define the state of the field at that time by summarising the current values held by devices as the partial map from device identifiers to values defined by $\phi(\delta) = \rho(\Psi(\delta)(\delta))$ if $\Psi(\delta)(\delta)$ exists. τ models *network topology*, namely, a directed neighbouring graph, as a map from device identifiers to set of identifiers. Σ models *sensor (distributed) state*, as a map from device identifiers to (local) sensors (i.e., sensor name/value maps). Then, *Env* (a couple of topology and sensor state) models the system's environment. So, a whole network configuration N is a couple of a status field and environment.

$F(\cdot)$ in Figure 10 (bottom), rule [N-FIR], models a filtering operation that clears out old stored values from the value-tree environments in Ψ, implicitly based on space/time tags. We use the following notation for status fields. Let $\overline{\delta} \mapsto \Theta$ denote the map sending each device identifier in $\overline{\delta}$ to the same value-tree environment Θ. Let $\Theta_0[\Theta_1]$ denote the value-tree environment with domain $\mathbf{dom}(\Theta_0) \cup \mathbf{dom}(\Theta_1)$ coinciding with Θ_1 in the domain of Θ_1 and with Θ_0 otherwise. Let $\Psi_0[\Psi_1]$ denote the status field with the *same domain* as Ψ_0 made of $\delta \mapsto \Psi_0(\delta)[\Psi_1(\delta)]$ for all δ in the domain of Ψ_1, $\delta \mapsto \Psi_0(\delta)$ otherwise.

We define network operational semantics in terms of small-steps transitions of the kind $N \xrightarrow{act} N'$, where *act* is either a device identifier in case it represents its firing, or label *env* to model any environment change. This is formalised in Figure 10 (bottom). Rule [N-FIR] models a computation round (firing) at device δ: it takes the local value-tree environment filtered out of old values $F(\Psi)(\delta)$; then by the single device semantics it obtains the device's value-tree θ, which is used to update the system configuration of δ's neighbours–the local sensors $\Sigma(\delta)$ are implicitly used by the auxiliary function $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}$ that gives the semantics to the built-in

functions. Rule [N-ENV] takes into account the change of the environment to a new well-formed environment Env'. Let $\overline{\delta}$ be the domain of Env'. We first construct a status field Ψ_0 associating to all the devices of Env' the empty context $\emptyset$. Then, we adapt the existing status field Ψ to the new set of devices: $\Psi_0[\Psi]$ automatically handles removal of devices, map of new devices to the empty context, and retention of existing contexts in the other devices.

Example 2. *In a possible implementation (by adding a "time tag" to every value tree t_θ and action label t_{act}) we can define $F(\Psi)$ as the mapping from $\overline{\delta}$ to $\overline{F(\Theta)}$ where*

$$F(\Theta) = \{\delta \mapsto \theta \in \Theta : \ t_\theta \geq t_{act} - t_{\mathtt{d}}\}$$

(recall that $t_{\mathtt{d}}$ is the decay parameter).

Furthermore, we can proceed in analogy with Example 1 and define a sequence of network evolution rules from a set of paths $\mathbf{P}$ together with a neighbouring predicate between positions and time tags for firings. In particular:

- *We introduce an occurrence of Rule [N-ENV] updating with the topology τ given by*

  $$\tau(\delta) = \{\delta' : \ neigh(\mathbf{P}(\delta)(t), \mathbf{P}(\delta')(t))\}$$

  for any time t corresponding to an activation change for a device (i.e. a border of an interval in which a path $\mathbf{P}(\delta)$ is defined);

- *We also introduce an occurrence of Rule [N-ENV] as above for any time t_ε corresponding to the firing ε of a device, each of them followed by*

- *an occurrence of Rule [N-FIR] on device δ_ε.*

The above sequence of rules is to be intended as sorted time-wise.

6 Properties of HFC

In this section we present the main properties of HFC, namely: 1) type preservation and domain alignment, intuitively meaning that HFC is "safe" in that it maintains lexical scoping in its handling of fields and computations with fields, and 2) adequacy and full abstraction, intuitively meaning that any aggregate-level program described within the denotational semantics is correctly implemented by the corresponding local actions of the operational semantics.

We shall defer all the proofs to Appendix A, focusing here on statements and definitions. We remark that throughout this paper we assume that the execution of every firing event ε terminates and is instantaneous and that every event happens at a distinct moment in time. Termination of the execution of a firing event can be enforced by means of several different techniques (see among others [23]), whose specific details fall outside the scope of this paper.

6.1 Device Computation Type Preservation and Domain Alignment

Here we formally state the device computation type preservation and domain alignment properties (cf. Section 3.4) for the HFC calculus. In order to state these properties we introduce the notion of well-formed value-tree environment for an expression.

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

- if $\mathtt{e}$ is a value, θ is of the form $\mathtt{v}\langle\rangle$;

- if $\mathtt{e} = \mathtt{nbr}\{\mathtt{e}_1\}$, θ is of the form $\mathtt{v}\langle\theta_1\rangle$;

- if $\mathtt{e} = \mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}_2\}$, θ is of the form $\mathtt{v}\langle\theta_1, \theta_2\rangle$ where θ_2 is well-formed for $\mathtt{e}_2$ with the additional assumption $\mathtt{x} : T$;

- if $\mathtt{e} = \mathtt{e}_{n+1}(\overline{\mathtt{e}})$, θ is of one of the following two forms:
  - $\mathtt{v}\langle\overline{\theta}, \theta_{n+1}\rangle$ where $\mathtt{f} = \rho(\theta_{n+1})$ is a built-in operator,
  - $\mathtt{v}\langle\overline{\theta}, \theta_{n+1}, \theta_{n+2}\rangle$ where $\mathtt{f}$ is not a built-in operator and θ_{n+2} is well-formed with respect to $\mathtt{e}_{n+2} = \mathit{body}(\mathtt{f})$ with the additional assumptions that $\mathit{args}(\mathtt{f}) : \overline{T}'$ where $\overline{\mathtt{e}} : \overline{T}'$.

In the above definition, θ_i is always assumed to be in the corresponding $\mathit{WFVT}(\mathscr{D}, \mathscr{A}, \mathtt{e}_i)$. The set $\mathit{WFVTE}(\mathscr{D}, \mathscr{A}, \mathtt{e})$ of the *well-formed value-tree environments* for $\mathtt{e}$ is such that $\Theta \in \mathit{WFVTE}(\mathscr{D}, \mathscr{A}, \mathtt{e})$ if and only if $\Theta = \overline{\delta} \mapsto \overline{\theta}$ where each θ_i is in $\mathit{WFVT}(\mathscr{D}, \mathscr{A}, \mathtt{e})$.

As these notions are defined we can now formally state the type preservation and domain alignment properties (cf. Section 3.4).

Theorem 1 (Device computation type preservation and domain alignment)**.** *Let $\mathscr{A} = \overline{\mathtt{x}} : \overline{T}$, $\mathscr{D}; \emptyset \vdash \overline{\mathtt{v}} : \overline{T}$, so that $\mathit{length}(\overline{\mathtt{v}}) = \mathit{length}(\overline{\mathtt{x}})$. If $\mathscr{D}; \mathscr{A} \vdash \mathtt{e} : T$, $\Theta \in \mathit{WFVTE}(\mathscr{D}, \mathscr{A}, \mathtt{e})$ and $\delta; \Theta \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] \Downarrow \theta$, then:*

1. *$\theta \in \mathit{WFVT}(\mathscr{D}, \mathscr{A}, \mathtt{e})$,*
2. *$\mathscr{D}; \emptyset \vdash \rho(\theta) : T$, and*
3. *if $\rho(\theta)$ is a neighbouring field value ϕ then $\boldsymbol{dom}(\phi) = \boldsymbol{dom}(\Theta) \cup \{\delta\}$.*

Proof. See Appendix A. □

6.2 Adequacy and full abstraction

We are now able to prove that the denotational semantics introduced in Section 4 is adequate and fully abstract with respect to the operational semantics introduced in Section 5. We shall prove these properties for monomorphic types, which will imply that given any parametric type *all of its possible instantiations* will satisfy the same property.

The notions of *adequacy* and *full-abstraction* are presented in literature in many (slightly) different forms (see e.g., [15, 42]), none of them fitting without modifications into the setting of HFC. In order to fix a reference, for an ML-like calculus we say that:

- The denotational semantics is *adequate* iff for any expression $\mathtt{e} : T$,
  - $\mathtt{e}$ converges operationally iff it converges denotationally,
  - if $\mathtt{e}$ evaluates to $\mathtt{v}$ then their denotations are equal ($[\![\mathtt{e}]\!] = [\![\mathtt{v}]\!]$),
  - and the converse of the last property holds whenever T is a built-in type.
- *Full abstraction* holds iff given any two expressions $\mathtt{e}_1, \mathtt{e}_2$ of a certain type T, their denotations coincide ($[\![\mathtt{e}_1]\!] = [\![\mathtt{e}_2]\!]$) if and only if for all contexts $\mathtt{e}'$ of built-in type with free variable $\mathtt{x} : T$, $\mathtt{e}'[\mathtt{x} := \mathtt{e}_1]$ and $\mathtt{e}'[\mathtt{x} := \mathtt{e}_2]$ evaluate to the same value.

Remark that the forward direction of full abstraction is implied by adequacy.

These notions need to be adjusted for the field calculus in order to accommodate for the influence of the environment and of the previous rounds of computations. In general, we achieve this roughly by adding a "for all environments" quantifier inside any operational or denotational statement.

Consider a program $\mathtt{e}_{\mathtt{main}}$ interpreted both operationally and denotationally, where each event ε is in bijection with one occurrence of rule [N-FIR]. Let Ψ_ε, τ_ε denote the status of the status field and topology just before the occurrence corresponding to ε. Let $\Theta_\varepsilon = F(\Psi_\varepsilon)(\delta_\varepsilon)$ denote the value-tree environment used in the computation of the firing corresponding to ε, and let θ_ε denote the outcome of this computation.

We say that the denotational environment is *coherent* with the operational evolution of a network if and only if:

1. For each ε in $\mathbf{E}$, $\Theta_\varepsilon = \{\delta_{\varepsilon'} \mapsto \theta_{\varepsilon'} : \mathit{neigh}(\varepsilon, \varepsilon')\}$. This is equivalent to the assertion that for each ε, ε' in $\mathbf{E}$, $\mathit{neigh}(\varepsilon, \varepsilon')$ holds if and only if:
   - $\delta_\varepsilon \in \tau_{\varepsilon'}(\delta_{\varepsilon'})$;

- there is no further ε'' between ε' and ε such that $\delta_{\varepsilon'} = \delta_{\varepsilon''}$ and $\delta_\varepsilon \in \tau_{\varepsilon''}(\delta_{\varepsilon''})$;
- $F(\Psi_\varepsilon)$ does not filter out ε' (referring to Example 2: $t_{\varepsilon'} \geq t_\varepsilon - t_{\mathtt{d}}$).

2. $\mathscr{B}[\![\mathtt{b}]\!]$ operates pointwise on its arguments (i.e. does not incorporate communication between events) and correctly translates the behaviour of $(\!|\mathtt{b}|\!)^{\Theta}_{\delta_\varepsilon}$, that is:

$$\mathscr{E}[\![(\!|\mathtt{b}|\!)^{\Theta}_{\delta_\varepsilon}(\overline{\mathtt{v}})]\!]^E = \mathscr{B}[\![\mathtt{b}]\!]\left(\mathscr{E}[\![\overline{\mathtt{v}}]\!]^E\right)$$

for all values $\overline{\mathtt{v}}$ of the correct type and E matching $\mathbf{dom}(\Theta)$.

We remark that the possible implementations outlined in Examples 1 and 2 are coherent.

Theorem 2 (Adequacy)**.** *Assume that the denotational environment is coherent with the operational evolution of the network and* $\mathtt{e}$ *is well-typed (that is,* $\mathscr{D};\emptyset \vdash \mathtt{e} : T$*).*

Then $\mathscr{E}[\![\mathtt{e}]\!]^{\mathbf{E}}(\varepsilon) = \mathscr{E}[\![\mathtt{v}_\varepsilon]\!]^{\mathbf{E}}(\varepsilon)$ *for each* ε *in* $\mathbf{E}$*, where* $\mathtt{v}_\varepsilon = \rho(\theta_\varepsilon)$ *is the operational outcome of fire* ε*.*

Proof. See Appendix A. □

We say that *full abstraction* holds for a field calculus iff given any two expression $\mathtt{e}_1$, $\mathtt{e}_2$ of a certain type T, $\mathscr{E}[\![\mathtt{e}_1]\!]^E = \mathscr{E}[\![\mathtt{e}_2]\!]^E$ for all E if and only if for all environments *Env* and contexts $\mathtt{e}'$ of built-in local type B with free variable $\mathtt{x} : T$, $\mathtt{e}'[\mathtt{x} := \mathtt{e}_1]$ and $\mathtt{e}'[\mathtt{x} := \mathtt{e}_2]$ evaluate to the same value trees in each firing.

The operational and denotational semantics hereby presented satisfy the full abstraction property, *provided that it holds for values of built-in local type*: we say that built-in constructors are faithful iff any two syntactically different expressions $\mathtt{c}(\overline{\ell})$, $\mathtt{c}'(\overline{\ell}')$ necessarily denote different objects.[8]

Theorem 3 (Full Abstraction)**.** *Suppose that constructors for built-in local types are faithful. Then the full abstraction property holds.*

Proof. See Appendix A. □

7 A Pervasive Computing Example

We now illustrate the application of field calculus, with a focus on first-class functions, using a pervasive computing example. In this scenario, people wandering a large environment (like an outdoor festival, an airport, or a museum) each carry a personal device with short-range point-to-point ad-hoc capabilities (e.g. a smartphone sending messages to others nearby via Bluetooth or Wi-Fi). All devices run a minimal "virtual machine" that allows runtime injection of new programs: any device can initiate a new distributed process (in the form of a 0-ary anonymous function), which the virtual machine spreads to all other devices within a specified range (e.g., 30 meters). For example, a person might inject a process that estimates crowd density by counting the number of nearby devices or a process that helps people to rendezvous with their friends, with such processes likely implemented via various self-organisation mechanisms. The virtual machine then executes these using the first-class function semantics above, providing predictable deployment and execution of an open class of runtime-determined processes.

7.1 Virtual Machine Implementation

The complete code for our example is listed in Figure 11. We use the following naming conventions for built-ins: functions `sns-*` embed sensors that return a value perceived from the environment (e.g., `sns-injection-point` returns a Boolean indicating whether a device's user wants to inject a function); functions `*-hood` yield a local value ℓ obtained by aggregating over the neighbouring field value ϕ in the input (e.g., `sum-hood` sums all values in each neighbourhood); functions `*-hood+` behave the same but exclude the value associated with the current device; and built-in functions `Pair`, `fst`, and `snd` respectively create a pair of locals and access a pair's first and second component.

[8] This property fails for example if we include constructors `Succ`, `Pred` for type `num`. On the other hand, it can hold for example if we assume to have a distinguished constructor n for every integer n.

```
;; Computes a field of minimum distance from 'source' devices
def distance-to (source) {   ;; has type: (bool) → num
  rep(infinity) { (d) => mux(source, 0, min-hood+( +[f,f](nbr{d}, nbr-range()))) }
}

;; Computes a field of pairs of distance to nearest 'source' device, and the most recent value of 'v' there
def gradcast (source, v) {    ;; has type: ∀ l. (bool, l) → l
  snd( (x) =>
          rep(x) {
            (t) => mux(source, Pair(0, v),
                        min-hood+(Pair[f,f](+[f,f]( nbr-range(), nbr{fst(t)}), nbr{snd(t)})))
          }
       (Pair(infinity, v)))
}

;; Evaluate a function field, running 'f' from 'source' within 'range' meters, and 'no-op' elsewhere
def deploy (range, source, g, no-op) {   ;; has type: ∀ l. (num, bool, () → l, () → l) → l
    if (distance-to(source) < range) {gradcast(source, g)} else {no-op} ()
}

;; The entry-point function executed to run the virtual machine on each device
def virtual-machine () {   ;; has type: () → num
    deploy( sns-range(), sns-injection-point(), sns-injected-fun(), () => 0)
}
```

```
;; Sums values of 'summand' into a minimum of 'potential', by descent
def converge-sum (potential, summand) {   ;; has type: (num, num) → num
  rep(summand) {
    (v) => summand +
           sum-hood+( mux[f,f,l]( nbr{parent(potential)} =[f,l] uid(), nbr{v}, 0))
  }
}

;; Maps each device to the uid of the neighbour with minimum value of 'potential'
def parent (potential) {   ;; has type: (num) → num
  snd( min-hood( Pair[l,f]( potential,
                            mux[f,f,l]( nbr{potential} <[f,l] potential, nbr{uid()}, NaN))))
}

;; Simple low-pass filter for smoothing noisy signal 'value' with rate constant 'alpha'
def low-pass (alpha, value) {   ;; has type: (num, num) → num
  rep(value) { (filtered) => *(value, alpha) + *(filtered, -(1, alpha)) }
}
```

Figure 11: Virtual machine code (top) and application-specific code (bottom).

The first two functions in Figure 11 implement frequently used self-organisation mechanisms. As already discussed, function `distance-to`, also known as *gradient* [13, 29], computes a field of minimal distances from each device to the nearest "source" device (those mapping to *true* in the Boolean input field). Note that the process of estimating distances self-stabilises into the desired field of distances, regardless of any transient perturbations or faults [28]. The second self-organisation mechanism, `gradcast`, is a directed broadcast, achieved by a computation identical to that of `distance-to`, except that the values are pairs (note that `Pair[f,f]` produces a neighbouring field of pairs, not a pair of neighbouring fields), with the second element set to the value of `v` at the source: `min-hood` operates on pairs by applying lexicographic ordering, so the second value of the pair is automatically carried along shortest paths from the source. The result is a field of pairs of distance and most recent value of `v` at the nearest source, of which only the value is returned.

The latter two functions in Figure 11 use these self-organisation methods to implement our simple virtual machine. Code mobility is implemented by function `deploy`, which spreads a 0-ary function `g` via `gradcast`, keeping it bounded within distance `range` from sources, and holding 0-ary function `no-op` elsewhere. The corresponding field of functions is then executed (note the double parenthesis). The `virtual-machine` then simply calls `deploy`, linking its arguments to sensors configuring deployment range and detecting who wants to inject which functions (and using `()=>0` as `no-op` function).

In essence, this virtual machine implements a code-injection model much like those used in a number of

other pervasive computing approaches (e.g., [31, 24, 9])—though of course it has much more limited features, since it is only an illustrative example. With these previous approaches, however, all code shares the same (global) lexical scope and cannot have its network domain externally controlled: this means that injected code may spread through the network unpredictably and may interact unpredictably with other injected code that it encounters. The extended field calculus semantics that we have presented, however, thanks to the restriction property (Section 4.5), ensures that injected code moves only within the range specified to the virtual machine and remains lexically isolated from different injected code, so that no variable can be unexpectedly affected by interactions with neighbours.

7.2 Simulation of Example Application

We further illustrate the example in a simulated scenario, considering a museum whose docents monitor their efficacy in part by tracking the number of patrons nearby while they are working. To monitor the number of nearby patrons, each docent's device injects the following anonymous function (of type: $() \to \texttt{num}$):

```
() => low-pass(0.5,converge-sum( distance-to(sns-injection-point()), mux(sns-patron(),1,0)))
```

This function is an anonymous version of the `track-count` function example in Section 2.2, using the same low-pass filtering of summation of a potential field to the docent, except that since the function cannot have any arguments, the Boolean fields indicating locations of patrons and docents are instead acquired via virtual sensors. In particular, in the `converge-sum` function, each device's local value is summed with those identifying it as their parent (their closest neighbour to the source, breaking ties with device unique identifiers from built-in function `uid`), resulting in a relatively balanced spanning tree of summations with the source at its root. This very simple version of summation is somewhat noisy on a moving network of devices [43], so its output is passed through a simple low-pass filter, the function `low-pass`, also defined in Figure 11 (bottom), in order to smooth its output and improve the quality of estimate.

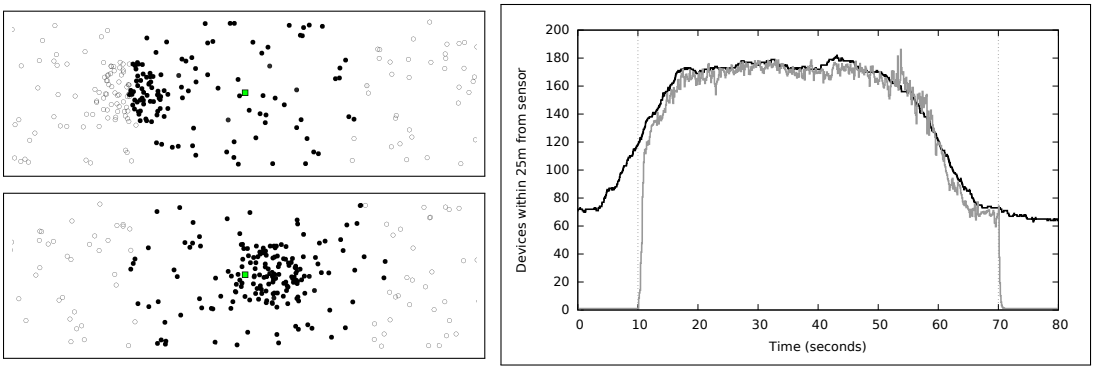
Figure 12a shows a simulation of a docent and 250 patrons in a large 100x30 meter museum gallery. Of the patrons, 100 are a large group of school-children moving together past the stationary docent from one side of the gallery to the other (thus causing a coherent rise and fall in local crowd density), while the rest are wandering randomly. In this simulation, people move at an average 1 m/s, the docent and all patrons carry personal devices running the virtual machine, executing asynchronously at 10Hz, and communicating via low-power Bluetooth to a range of 10 meters—hence, hop-by-hop communication is needed for longer range interaction. The simulation was implemented using the ALCHEMIST [39] simulation framework and the Protelis [40] incarnation of field calculus, updated to the extended version of the calculus presented in this paper.

In this simulation, at time 10 seconds, the docent injects the patron-counting function with a range of 25 meters, and at time 70 seconds removes it. Figure 12a shows two snapshots of the simulation, at times 11 (top) and 35 (bottom) seconds, while Figure 12b compares the estimated value returned by the injected process with the true value. Note that upon injection, the process rapidly disseminates and begins producing good estimates of the number of nearby patrons, then cleanly terminates upon removal.

All together, these examples illustrate how the coherence between global denotational and local operational semantics in field calculus allows complex distributed applications to be safely and elegantly implemented with quite compact code, including higher-order operations like process management and runtime deployment of applications in an open environment.

8 Conclusion and Future Work

Conceiving emerging distributed systems in terms of computations involving aggregates of devices, and hence adopting higher-level abstractions for system development, is a thread that has recently received a good deal of attention, as discussed in Section 2. Those that best support self-organisation approaches to robust and environment-independent computations, however, have generally lacked well-engineered mechanisms to support openness and code mobility (injection, update, etc.). Our contribution has been to develop a core calculus, building on the work presented in [46], that for the first time smoothly combines self-organisation



(a) Simulation snapshots

(b) Estimated vs. True Count

Figure 12: (a) Two snapshots of museum simulation: patrons (grey) are counted (black) within 25 meters of the docent (green). (b) Estimated number of nearby patrons (grey) vs. actual number (black) in the simulation, during the period where the docent is running the patron counting function (between two dashed lines).

and code mobility, by means of the abstraction of a "distributed function field". This combination of first-class functions with the domain-restriction mechanisms of field calculus allows the predictable and safe composition of distributed self-organisation mechanisms at runtime, thereby enabling robust operation of open pervasive systems. Furthermore, the simplicity of the calculus enables it to easily serve as both an analytical framework and a programming framework, and we have already incorporated this into Protelis [40], thereby allowing these mechanisms to be deployed both in simulation and in actual distributed systems.

Future plans include consolidation of this work, by extending the calculus and its conceptual framework, to support an analytical methodology and a practical toolchain for system development, as outlined in [6] and [5]. We aim to apply our approach to support various application needs for dynamic management of distributed processes [2], which may also impact the methods of alignment for anonymous functions. We also plan to isolate fragments of the calculus that satisfy behavioural properties such as self-stabilisation, quasi-stabilisation to a dynamically evolving field, or density independence, following the approach of [45] and [43]. Finally, these foundations can be applied in developing APIs enabling the simple construction of complex distributed applications, building on the work in [6], [5] and [43] to define a layered library of self-organisation patterns, and applying these APIs to support a wide range of practical distributed applications.

A Appendix: Proofs of the Main Theorems

A.1 Device Computation Type Preservation and domain Alignment

Restatement of Theorem 1 (Type Preservation and Domain Alignment). *Let* $\mathscr{A} = \overline{\mathtt{x}} : \overline{T}$, $\mathscr{D};\emptyset \vdash \overline{\mathtt{v}} : \overline{T}$, *so that* $length(\overline{\mathtt{v}}) = length(\overline{\mathtt{x}})$. *If* $\mathscr{D};\mathscr{A} \vdash \mathtt{e} : T$, $\Theta \in WFVTE(\mathscr{D},\mathscr{A},\mathtt{e})$ *and* $\delta;\Theta \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] \Downarrow \theta$, *then:*

1. $\theta \in WFVT(\mathscr{D},\mathscr{A},\mathtt{e})$,
2. $\mathscr{D};\emptyset \vdash \rho(\theta) : T$, *and*
3. *if* $\rho(\theta)$ *is a neighbouring field value* ϕ *then* $\boldsymbol{dom}(\phi) = \boldsymbol{dom}(\Theta) \cup \{\delta\}$.

Observe that the typing rules (in Figure 4) and the evaluation rules (in Figure 9) are syntax directed. Then the proof can be carried out by induction on the syntax of expressions, while using the following standard lemmas.

Lemma 4 (Substitution)**.** *Let* $\mathscr{A} = \overline{\mathtt{x}} : \overline{T}$, $\mathscr{B};\emptyset \vdash \overline{\mathtt{v}} : \overline{T}$. *If* $\mathscr{D};\mathscr{A} \vdash \mathtt{e} : T$, *then* $\mathscr{D};\emptyset \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] : T$.

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

Lemma 5 (Weakening)**.** *Let $\mathscr{D}' \supseteq \mathscr{D}$, $\mathscr{A}' \supseteq \mathscr{A}$ be such that $\boldsymbol{dom}(\mathscr{D}') \cap \boldsymbol{dom}(\mathscr{A}') = \emptyset$. If $\mathscr{D};\mathscr{A} \vdash \mathtt{e} : T$, then $\mathscr{D}';\mathscr{A}' \vdash \mathtt{e} : T$.*

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

of Theorem 1. We proceed proving points (1)-(3) by simultaneous induction on the syntax of expression $\mathtt{e}$ (given in Figure 2).

- $\mathtt{e} = \phi$: This case is not allowed to appear in source programs.
- $\mathtt{e} = \mathtt{c}(\overline{\ell}) \mid \mathtt{b} \mid \mathtt{d} \mid (\overline{\mathtt{x}})$ `=>` $\mathtt{e}$: In this case, $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] = \mathtt{v}$ is a local value hence $\theta = \mathtt{v}\langle\rangle$ by rule [E-LOC]. Thus θ is well-formed for $\mathtt{e}$ and $\rho(\theta) = \mathtt{v}$ has type T by the Substitution Lemma.
- $\mathtt{e} = \mathtt{x}$: In this case, $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]$ is trivially a value of the correct type T, hence $\theta = \mathtt{v}\langle\rangle$ is well-formed for $\mathtt{e}$. If T is a local type, we are done. If T is a field type, we know by induction hypothesis that $\phi = \mathtt{v}$ has domain $\mathbf{dom}(\Theta') \cup \{\delta\}$ for some Θ' including Θ as a subtree (pointwise). Thus $\mathbf{dom}(\phi) \supseteq \mathbf{dom}(\Theta) \cup \{\delta\}$, hence we can apply rule [E-FLD] to obtain that ϕ has domain exactly $\mathbf{dom}(\Theta) \cup \{\delta\}$.
- $\mathtt{e} = \mathtt{e}_{n+1}(\overline{\mathtt{e}})$: In this case, either rule [E-B-APP] or [E-D-APP] applies, depending on whether $\mathtt{e}_{n+1}$ evaluates to a built-in operator or not. In both cases, the resulting value-tree θ is easily checked to be well-formed for $\mathtt{e}$, and type preservation holds by induction hypothesis together with standard arguments on typing of function applications.

  If $\mathtt{e}$ has field type, we also need to check that $\mathbf{dom}(\rho(\theta)) = \mathbf{dom}(\Theta) \cup \{\delta\}$. We have two possibilities:

  - $\mathtt{e}_{n+1}$ evaluates to a built-in operator $\mathtt{b}$: In this case, domain alignment follows from the assumptions on $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}$.
  - $\mathtt{e}_{n+1}$ evaluates to a user-defined or anonymous function $\mathtt{f}$: In this case, $\mathtt{e}_{n+1}$ cannot be of the form $\mathtt{rep}$, $\mathtt{nbr}$, or $\mathtt{c}(\ldots)$. Furthermore, it cannot neither be of the form $\mathtt{e}'(\overline{\mathtt{e}}')$ since in that case $\mathtt{e}'$ would have type $(\overline{T}') \to (\overline{T}) \to F$ which is not allowed by the type system. It follows that $\mathtt{e}_{n+1}$ is in fact equal to $\mathtt{f}$, hence no alignment with neighbours is required.
- $\mathtt{e} = \mathtt{nbr}\{\mathtt{e}_1\}$: In this case rule [E-NBR] applies, thus $\theta = \phi\langle\theta_1\rangle$ where $\phi = \rho(\pi_1(\Theta))[\delta \mapsto \rho(\theta_1)]$. Then θ is well-formed for $\mathtt{e}$, and ϕ has type $T = \mathtt{field}(T_1)$ where $\mathtt{e}_1 : T_1$. Furthermore, $\mathbf{dom}(\phi) = \mathbf{dom}(\Theta) \cup \{\delta\}$ as required.
- $\mathtt{e} = \mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})$ `=>` $\mathtt{e}_2\}$: In this case rule [E-REP] applies, thus $\theta = \ell_2\langle\theta_1, \theta_2\rangle$ is well-formed for $\mathtt{e}$ (where ℓ_i, θ_i follows the notation in Figure 9, rule [E-REP]). By induction hypothesis, $\ell_1 = \rho(\theta_1)$ has the same type as $\mathtt{e}_1$ which is T. If $\delta \notin \mathbf{dom}(\Theta)$, $\ell_0 = \ell_1$ also has type T. Otherwise, since Θ is well-formed for $\mathtt{e}$, $\pi_2(\Theta)$ is well-formed for $\mathtt{e}_2$ with the additional assumption that $\mathtt{x} : T$, and by induction hypothesis $\ell_0 = \rho(\pi_2(\Theta))(\delta)$ has the same type as $\mathtt{e}_2$ which is also T. Then by the Substitution Lemma, $\mathtt{e}_2[\mathtt{x} := \ell_0]$ also has type T hence by induction hypothesis the same does $\ell_2 = \rho(\theta_2)$, concluding the proof.

□

A.2 Adequacy and full abstraction

Restatement of Theorem 2 (Adequacy). *Assume that the denotational environment is coherent with the operational evolution of the network and $\mathtt{e}$ is well-typed (that is, $\mathscr{D};\emptyset \vdash \mathtt{e} : T$).*

Then $\mathscr{E}[\![\mathtt{e}]\!]^{\mathbf{E}}(\varepsilon) = \mathscr{E}[\![\mathtt{v}_\varepsilon]\!]^{\mathbf{E}}(\varepsilon)$ for each ε in $\mathbf{E}$, where $\mathtt{v}_\varepsilon = \rho(\theta_\varepsilon)$ is the operational outcome of fire ε.

In order to carry on the induction on the structure of $\mathtt{e}$, we shall prove the following strengthened version instead.

Lemma 6. *Assume that the denotational environment is coherent with the operational evolution of the network and $\mathtt{e}$ is a well-typed expression with free variables $\overline{\mathtt{x}}$ (that is, $\mathscr{D};\overline{\mathtt{x}} : \overline{T} \vdash \mathtt{e} : T$). Let $X = \overline{\mathtt{x}} \mapsto \overline{\Phi}$ and $\overline{\mathtt{u}}^{\varepsilon}$ be such that $\Phi_i(\varepsilon) = \mathscr{E}[\![\mathtt{u}_i^{\varepsilon}]\!](\varepsilon)$ for all i and events ε.*

Then $\mathscr{E}[\![\mathtt{e}]\!]_X(\varepsilon) = \mathscr{E}[\![\mathtt{v}^{\varepsilon}]\!](\varepsilon)$ for each ε in $\mathbf{E}$, where $\mathtt{v}^{\varepsilon} = \rho(\theta_\varepsilon)$ is the operational outcome of fire ε evaluating $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{u}}^{\varepsilon}]$.

Proof. First, recall that coherence of denotational and operational environments implies that for each fire ε, $\Theta_\varepsilon = \{\delta_{\varepsilon'} \mapsto \theta_{\varepsilon'} : \ neigh(\varepsilon, \varepsilon')\}$. We prove the assertion simultaneously for all possible (pairs of coherent) environments, set of assumptions and event ε, by induction on the structure of $\mathtt{e}$.

- $\mathtt{e} = \mathtt{x}_i$: In this case, $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{u}}^\varepsilon] = \mathtt{u}_i^\varepsilon = \mathtt{v}^\varepsilon$ and by hypothesis $\Phi_i(\varepsilon) = \mathscr{E}[\![\mathtt{u}_i^\varepsilon]\!](\varepsilon) = \mathscr{E}[\![\mathtt{v}^\varepsilon]\!](\varepsilon)$.
- $\mathtt{e} = \mathtt{c}(\overline{\ell}) \mid \phi \mid \mathtt{b} \mid \mathtt{d} \mid (\overline{\mathtt{x}})\ \mathtt{=>}\ \mathtt{e}$: Since $\mathtt{e}$ is already a value, $\mathtt{v}^\varepsilon = \mathtt{e}$ and the thesis follows.
- $\mathtt{e} = \mathtt{nbr}\{\mathtt{e}_1\}$: By inductive hypothesis $\mathscr{E}[\![\mathtt{e}_1]\!]_X(\varepsilon) = \mathscr{E}[\![\mathtt{v}_1^\varepsilon]\!](\varepsilon)$ for each ε in $\mathbf{E}$, where $\mathtt{v}_1^\varepsilon = \rho(\theta_1^\varepsilon)$ is the (operational) outcome of $\mathtt{e}_1[\overline{\mathtt{x}} := \overline{\mathtt{u}}^\varepsilon]$ in fire ε. By rule [E-NBR], we have that

$$\mathtt{v}^\varepsilon = \{\delta_{\varepsilon'} \mapsto \mathtt{v}_1^{\varepsilon'} : \ (\varepsilon' = \varepsilon) \vee (neigh(\varepsilon, \varepsilon') \wedge \delta_{\varepsilon'} \neq \delta_\varepsilon)\}$$

  and

$$\mathscr{E}[\![\mathtt{v}^\varepsilon]\!](\varepsilon) = \{\delta_{\varepsilon'} \mapsto \mathscr{E}[\![\mathtt{v}_1^{\varepsilon'}]\!](\varepsilon) : \ (\varepsilon' = \varepsilon) \vee (neigh(\varepsilon, \varepsilon') \wedge \delta_{\varepsilon'} \neq \delta_\varepsilon)\}.$$

  On the other hand, $\mathscr{E}[\![\mathtt{e}]\!]_X(\varepsilon)$ is

$$\begin{aligned}
\mathscr{E}[\![\mathtt{nbr}\{\mathtt{e}_1\}]\!]_X(\varepsilon) &= \lambda\delta \in E^-(\varepsilon).\mathscr{E}[\![\mathtt{e}_1]\!]_X(\varepsilon^\delta) \\
&= \lambda\delta \in E^-(\varepsilon).\mathscr{E}[\![\mathtt{v}_1^{\varepsilon^\delta}]\!](\varepsilon^\delta) \\
&= \{\delta \mapsto \mathscr{E}[\![\mathtt{v}_1^{\varepsilon^\delta}]\!](\varepsilon) : \ (\varepsilon^\delta = \varepsilon) \vee (neigh(\varepsilon, \varepsilon^\delta) \wedge \delta \neq \delta_\varepsilon)\} = \mathscr{E}[\![\mathtt{v}^\varepsilon]\!](\varepsilon).
\end{aligned}$$

- $\mathtt{e} = \mathtt{rep}(\mathtt{e}_1)\{(\mathtt{y})\mathtt{=>}\mathtt{e}_2\}$: Recall that by rule [E-REP], $\mathtt{v}^\varepsilon$ is the evaluation of $\mathtt{e}_2[\overline{\mathtt{x}} := \overline{\mathtt{u}}^\varepsilon, \mathtt{y} := \mathtt{v}^{\varepsilon^-}]$ if ε^- exists, $\mathtt{e}_2[\overline{\mathtt{x}} := \overline{\mathtt{u}}^\varepsilon, \mathtt{y} := \mathtt{v}_1^\varepsilon]$ otherwise. We prove by induction on n that if ε has at most n predecessors (i.e. ε^- can be applied at most n times) then $\mathscr{E}[\![\mathtt{v}^\varepsilon]\!](\varepsilon) = \mathscr{R}[\![\mathtt{e}]\!]_{n+1}(\varepsilon)$. The thesis will then follow for n greater than the number of events.

  If $n = 0$, by induction hypothesis on both $\mathtt{e}_1$ and $\mathtt{e}_2[\overline{\mathtt{x}} := \overline{\mathtt{u}}^\varepsilon, \mathtt{y} := \mathtt{v}_1^\varepsilon]$:

$$\mathscr{E}[\![\mathtt{v}^\varepsilon]\!](\varepsilon) = \mathscr{E}[\![\mathtt{e}_2]\!]_{X \cup \mathtt{y} \mapsto \mathscr{E}[\![\mathtt{e}_1]\!]_X}(\varepsilon) = \mathscr{R}[\![\mathtt{e}]\!]_1(\varepsilon).$$

  If $n > 0$, $\mathtt{v}^{\varepsilon^-}$ has at most $n-1$ predecessors thus we can apply the inductive hypothesis on $n-1$ and $\mathtt{e}_2[\overline{\mathtt{x}} := \overline{\mathtt{u}}^\varepsilon, \mathtt{y} := \mathtt{v}^{\varepsilon^-}]$ to obtain:

$$\mathscr{E}[\![\mathtt{v}^\varepsilon]\!](\varepsilon) = \mathscr{E}[\![\mathtt{e}_2]\!]_{X \cup \mathtt{y} \mapsto \mathbf{shift}(\mathscr{R}[\![\mathtt{e}]\!]_n)}(\varepsilon) = \mathscr{R}[\![\mathtt{e}]\!]_{n+1}(\varepsilon).$$

- $\mathtt{e} = \mathtt{e}_{n+1}(\overline{\mathtt{e}})$: Suppose that $\overline{\mathtt{e}}$ evaluates to $\overline{\mathtt{v}}^\varepsilon$ and $\mathtt{e}_{n+1}$ to $\mathtt{f}^\varepsilon$ in event ε. Notice that by induction hypothesis, $\mathscr{E}[\![\overline{\mathtt{v}}^\varepsilon]\!](\varepsilon) = \mathscr{E}[\![\overline{\mathtt{e}}]\!]_X(\varepsilon)$ and $\mathscr{E}[\![\mathtt{f}^\varepsilon]\!](\varepsilon) = \mathscr{E}[\![\mathtt{e}_{n+1}]\!]_X(\varepsilon)$.

  If $\mathtt{f}^\varepsilon$ is a built-in operator $\mathtt{b}$, the thesis follows from coherence of $\mathscr{B}[\![\mathtt{b}]\!]$ with $(\!|\mathtt{b}|\!)^\Theta_\delta$, together with the induction hypothesis on $\mathtt{e}_1, \ldots, \mathtt{e}_{n+1}$.

  If $\mathtt{f}^\varepsilon$ is an anonymous function, by rule [E-D-APP] $\mathtt{f}^\varepsilon(\overline{\mathtt{v}}^\varepsilon)$ (hence $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{u}}^\varepsilon]$) evaluates in ε to the result $\mathtt{v}^\varepsilon$ of $body(\mathtt{f}^\varepsilon)[args(\mathtt{f}^\varepsilon) := \overline{\mathtt{v}}^\varepsilon]$ as calculated in $\Theta'_\varepsilon = \pi^{\mathtt{f}_\varepsilon}(\Theta_\varepsilon)$. The value-tree environments $\{\Theta'_{\varepsilon'} : \ \mathtt{f}^{\varepsilon'} = \mathtt{f}^\varepsilon\}$ together define an operational environment $Env_{\mathtt{f}^\varepsilon}$ consisting only of those devices and events which agree on the evaluation of $\mathtt{e}_{n+1}$. In fact, such restricted environment is coherent with the restricted denotational environment $\mathbf{E}(\mathtt{e}_{n+1}, \varepsilon)$:

$$\begin{aligned}
\mathbf{E}(\mathtt{e}_{n+1}, \varepsilon) &= \{\varepsilon' : \ \mathscr{E}[\![\mathtt{e}_{n+1}]\!]_X(\varepsilon') = \mathscr{E}[\![\mathtt{e}_{n+1}]\!]_X(\varepsilon)\} \\
&= \{\varepsilon' : \ \mathscr{E}[\![\mathtt{f}^{\varepsilon'}]\!](\varepsilon') = \mathscr{E}[\![\mathtt{f}^\varepsilon]\!](\varepsilon)\} \\
&= \{\varepsilon' : \ \mathscr{E}[\![\mathtt{f}^{\varepsilon'}]\!] = \mathscr{E}[\![\mathtt{f}^\varepsilon]\!]\} = \{\varepsilon' : \ \mathtt{f}^{\varepsilon'} = \mathtt{f}^\varepsilon\}
\end{aligned}$$

  where we used the inductive hypothesis and the facts that denotations of values are constant field evolutions and that function denotations coincide if and only if the functions are syntactically equal (in order).

Thus we can apply the induction hypothesis on $\overline{\mathtt{e}}$ in $\mathbf{E}$ and on $\mathit{body}(\mathtt{f}^{\varepsilon})[\mathit{args}(\mathtt{f}^{\varepsilon}) := \overline{\mathtt{v}}^{\varepsilon}]$ in $\mathbf{E}(\mathtt{e}_{n+1}, \varepsilon)$ to obtain:

$$\mathscr{E}[\![\mathtt{v}^{\varepsilon}]\!](\varepsilon) = \mathscr{E}[\![\mathit{body}(\mathtt{f}^{\varepsilon})]\!]^{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}_{\mathit{args}(\mathtt{f}^{\varepsilon})\mapsto\mathscr{E}[\![\overline{\mathtt{e}}]\!]^{\mathbf{E}}_{X}|_{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}}(\varepsilon)$$

On the other hand,

$$\begin{aligned}
\mathscr{E}[\![\mathtt{e}]\!]_X(\varepsilon) &= \mathtt{snd}\left(\mathscr{E}[\![\mathtt{e}_{n+1}]\!]_X(\varepsilon)\right)\left(\mathscr{E}[\![\overline{\mathtt{e}}]\!]_X|_{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}\right)(\varepsilon)\\
&= \mathtt{snd}\left(\mathscr{E}[\![\mathtt{f}^{\varepsilon}]\!](\varepsilon)\right)\left(\mathscr{E}[\![\overline{\mathtt{e}}]\!]_X|_{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}\right)(\varepsilon)\\
&= \left(\lambda\overline{\Phi}.\mathscr{E}[\![\mathit{body}(\mathtt{f}^{\varepsilon})]\!]^{\mathbf{dom}(\overline{\Phi})}_{\mathit{args}(\mathtt{f}^{\varepsilon})\mapsto\overline{\Phi}}\right)\left(\mathscr{E}[\![\overline{\mathtt{e}}]\!]_X|_{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}\right)(\varepsilon)\\
&= \mathscr{E}[\![\mathit{body}(\mathtt{f}^{\varepsilon})]\!]^{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}_{\mathit{args}(\mathtt{f}^{\varepsilon})\mapsto\mathscr{E}[\![\overline{\mathtt{e}}]\!]^{\mathbf{E}}_{X}|_{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}}(\varepsilon) = \mathscr{E}[\![\mathtt{v}^{\varepsilon}]\!](\varepsilon)
\end{aligned}$$

completing the proof in this case.

If $\mathtt{f}^{\varepsilon}$ is a user-defined function, we prove by further induction on n that $\mathscr{E}[\![\mathtt{v}^{\varepsilon}]\!](\varepsilon)$ is equal to $V_n = \mathscr{E}[\![\mathit{body}(\mathtt{f}^{\varepsilon})]\!]^{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}_{\mathit{args}(\mathtt{f}^{\varepsilon})\mapsto\mathscr{E}[\![\overline{\mathtt{e}}]\!]^{\mathbf{E}}_{X}|_{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)},\mathtt{f}^{\varepsilon}\mapsto\mathscr{D}[\![\mathtt{f}^{\varepsilon}]\!]_n}(\varepsilon)$ whenever the recursive depth of the function call is bounded by n. The thesis will then follow by passing to the limit (and expanding the denotation of $\mathtt{e}$ as for anonymous functions).

If $n = 0$, the evaluation of $\mathit{body}(\mathtt{f}^{\varepsilon})[\mathit{args}(\mathtt{f}^{\varepsilon}) := \overline{\mathtt{v}}^{\varepsilon}]$ does not involve further evaluation of the defined function $\mathtt{f}^{\varepsilon}$. This also holds on the denotational side, giving that

$$V_0 = \mathscr{E}[\![\mathit{body}(\mathtt{f}^{\varepsilon})]\!]^{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}_{\mathit{args}(\mathtt{f}^{\varepsilon})\mapsto\mathscr{E}[\![\overline{\mathtt{e}}]\!]^{\mathbf{E}}_{X}|_{\mathbf{E}(\mathtt{e}_{n+1},\varepsilon)}}(\varepsilon) = \mathscr{E}[\![\mathtt{v}^{\varepsilon}]\!](\varepsilon)$$

If instead $n > 0$, the evaluation of $\mathit{body}(\mathtt{f}^{\varepsilon})[\mathit{args}(\mathtt{f}^{\varepsilon}) := \overline{\mathtt{v}}^{\varepsilon}]$ involves evaluation of $\mathtt{f}^{\varepsilon}$ with recursive depth bounded by $n-1$. Thus by inductive hypothesis we can denote each such call with $\mathscr{D}[\![\mathtt{f}^{\varepsilon}]\!]_{n-1}$ and the thesis follows.

□

Restatement of Theorem 3 (Full Abstraction). *Suppose that constructors for built-in local types are faithful. Then the full abstraction property holds.*

Proof. First notice that given any two values $\mathtt{v}_1$, $\mathtt{v}_2$ of the same type $\mathscr{E}[\![\mathtt{v}_1]\!]^E = \mathscr{E}[\![\mathtt{v}_2]\!]^E$ if and only if $\mathtt{v}_1 = \mathtt{v}_2$. If $\mathtt{v}_1$, $\mathtt{v}_2$ are functions, it holds since the denotation includes the function tag (i.e. the syntactic expression itself). If $\mathtt{v}_1$, $\mathtt{v}_2$ are constructor expressions, it holds by hypothesis. If $\mathtt{v}_1$, $\mathtt{v}_2$ are neighbouring field values, it holds since $\mathscr{E}[\![\overline{\delta} \mapsto \overline{\ell}]\!]^E = \lambda\varepsilon.\overline{\delta} \mapsto \mathscr{E}[\![\overline{\ell}]\!]^E(\varepsilon)$ and we already proved the equivalence for local values. From this equivalence and adequacy we can prove the left-to-right implication of the full abstraction property as follows.

Suppose that $\mathtt{e}_1$ and $\mathtt{e}_2$ have the same denotation in every denotational environment and fix an operational environment *Env*, in which $\mathtt{e}_1$, $\mathtt{e}_2$ evaluate to $\mathtt{v}^1_{\varepsilon}$, $\mathtt{v}^2_{\varepsilon}$ in fire ε. Then given a denotational environment E that is coherent with *Env*, we have that $\mathscr{E}[\![\mathtt{e}_1]\!]^E = \mathscr{E}[\![\mathtt{e}_2]\!]^E$. By adequacy, it follows that $\mathscr{E}[\![\mathtt{v}^1_{\varepsilon}]\!]^E(\varepsilon) = \mathscr{E}[\![\mathtt{v}^2_{\varepsilon}]\!]^E(\varepsilon)$ hence $\mathtt{v}^1_{\varepsilon} = \mathtt{v}^2_{\varepsilon}$ for all ε. Since $\mathtt{e}_1$ and $\mathtt{e}_2$ evaluate to the same value in every fire ε they cannot be distinguished in any context in *Env*. Since *Env* was arbitrary, this concludes the left-to-right part of the proof.

For the converse implication, suppose that there exists a denotational environment E such that $\mathtt{e}_1$ and $\mathtt{e}_2$ have denotations which differs in ε. Let *Env* be an operational environment coherent with E, in which $\mathtt{e}_1$, $\mathtt{e}_2$ evaluate to $\mathtt{v}_1$, $\mathtt{v}_2$ in fire ε. By adequacy,

$$\mathscr{E}[\![\mathtt{v}_1]\!]^E(\varepsilon) = \mathscr{E}[\![\mathtt{e}_1]\!]^E(\varepsilon) \neq \mathscr{E}[\![\mathtt{e}_2]\!]^E(\varepsilon) = \mathscr{E}[\![\mathtt{v}_2]\!]^E(\varepsilon)$$

hence $\mathtt{v}_1 \neq \mathtt{v}_2$. Thus $\mathtt{e}_1$, $\mathtt{e}_2$ are distinguished by context $\mathtt{e}' ::= (\mathtt{e}_1 = \mathtt{x})$ of type $\mathtt{bool}$.

□

References

[1] Jos CM Baeten, Twan Basten, Twan Basten, and MA Reniers. *Process algebra: equational theories of communicating processes*, volume 50. Cambridge university press, 2010.

[2] Jacob Beal. Dynamically defined processes for spatial computers. In *Spatial Computing Workshop*, pages 206–211, New York, September 2009. IEEE.

[3] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, March/April 2006.

[4] 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`.

[5] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the Internet of Things. *IEEE Computer*, 48(9), 2015.

[6] Jacob Beal and Mirko Viroli. Building blocks for aggregate programming of self-organising applications. In *2nd FoCAS Workshop on Fundamentals of Collective Systems*, pages 1–6. IEEE CS, to appear, 2014.

[7] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution changes in situated computing systems. In *IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO 2016)*. IEEE, 2016. To appear.

[8] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gian Luigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The Klaim project: Theory and practice. In *Global Computing 2003*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer, 2003.

[9] William Butera. *Programming a Paintable Computer*. PhD thesis, MIT, Cambridge, USA, 2002.

[10] Luca Cardelli and Philippa Gardner. Processes in space. In *6th Conference on Computability in Europe*, volume 6158 of *Lecture Notes in Computer Science*, pages 78–87. Springer, 2010.

[11] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, June 2000.

[12] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[13] Lauren Clement and Radhika Nagpal. Self-assembly and self-repairing topologies. In *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*, 2003.

[14] Daniel Coore. *Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer*. PhD thesis, MIT, Cambridge, MA, USA, 1999.

[15] Pierre-Louis Curien. Definability and full abstraction. *Electr. Notes Theor. Comput. Sci.*, 172:301–310, 2007.

[16] 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.

[17] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. A type-sound calculus of computational fields. *Science of Computer Programming*, 117:17 – 44, 2016.

[18] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. Code mobility meets self-organisation: A higher-order calculus of computational fields. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 9039 of *Lecture Notes in Computer Science*, pages 113–128. Springer International Publishing, 2015.

[19] J. W. de Bakker and Jeffery I. Zucker. Denotational semantics of concurrency. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 153–158, 1982.

[20] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. A language-based approach to autonomic computing. In *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48, 2013.

[21] Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluís Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, 12(1):43–67, 2013.

[22] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, CIKM '94, pages 456–463, New York, NY, USA, 1994. ACM.

[23] William I. Gasarch. Proving programs terminate using well-founded orderings, ramsey's theorem, and matrices. *Advances in Computers*, 97:147–200, 2015.

[24] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[25] Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Przemyslaw Prusinkiewicz. Computational models for integrative and developmental biology. Technical Report 72-2002, Univerite d'Evry, LaMI, 2002.

[26] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.

[27] M.E. Inchiosa and M.T. Parker. Overcoming design and development challenges in agent-based modeling using ascape. *Proceedings of the National Academy of Sciences of the United States of America*, 99(Suppl 3):7304, 2002.

[28] Shay Kutten and Boaz Patt-Shamir. Time-adaptive self stabilization. In *Proceedings of ACM symposium on Principles of distributed computing*, pages 149–158. ACM, 1997.

[29] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Trans. Softw. Eng.*, 13(1):32–38, 1987.

[30] Samuel R. Madden, Robert Szewczyk, Michael J. Franklin, and David Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[31] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. on Software Engineering Methodologies*, 18(4):1–56, 2009.

[32] José Meseguer, Joseph A Goguen, and Gert Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8(4):383–413, 1989.

[33] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2*, September 2009.

[34] Robin Milner. Pure bigraphs: Structure and dynamics. *Information and Computation*, 204(1):60 – 122, 2006.

[35] Radhika Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, Cambridge, MA, USA, 2001.

[36] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Workshop on Data Management for Sensor Networks*, pages 78–87, August 2004.

[37] Yuichi Nishiwaki. Digamma-calculus: A universal programming language of self-stabilizing computational fields. In *Workshop eCAS in Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2016 IEEE International Conference on*. IEEE, Sept 2016.

[38] Atsushi Ohori. A simple semantics for ml polymorphism. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 281–292. ACM, 1989.

[39] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with Alchemist. *Journal of Simulation*, 7:202–215, 2013.

[40] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: Practical aggregate programming. In *ACM Symposium on Applied Computing 2015*, pages 1846–1853, April 2015.

[41] E. Sklar. Netlogo, a multi-agent simulation environment. *Artificial life*, 13(3):303–311, 2007.

[42] Allen Stoughton. *Fully abstract models of programming languages*. Research Notes in Theoretical Computer Science. Pitman, 1988.

[43] 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.

[44] Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems*, 6(2):14:1 – 14:24, June 2011.

[45] Mirko Viroli and Ferruccio Damiani. A calculus of self-stabilising computational fields. In *Coordination Languages and Models*, volume 8459 of *LNCS*, pages 163–178. Springer-Verlag, June 2014.

[46] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg, 2013.

[47] Mirko Viroli, Danilo Pianini, and Jacob Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *Proceedings of Coordination 2012*, volume 7274 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2012.

[48] Mirko Viroli, Danilo Pianini, Sara Montagna, Graeme Stevenson, and Franco Zambonelli. A coordination model of pervasive service ecosystems. *Science of Computer Programming*, 110:3 – 22, 2015.

[49] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004.

[50] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.