

A Higher-Order Calculus of Computational Fields

GIORGIO AUDRITO, Dipartimento di Informatica, University of Turin, Italy

MIRKO VIROLI, Dipartimento di Informatica – Scienza e Ingegneria, Università di Bologna, Italy

FERRUCCIO DAMIANI, Dipartimento di Informatica, University of Turin, Italy

DANILO PIANINI, Dipartimento di Informatica – Scienza e Ingegneria, Università di Bologna, Italy

JACOB BEAL, Raytheon BBN Technologies, USA

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 article 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 computational adequacy and abstraction for the (aggregate) denotational semantics with respect to the (per-device) operational semantics.

CCS Concepts: • **Theory of computation** → **Distributed computing models**; **Process calculi**; **Type structures**; **Denotational semantics**; **Operational semantics**; *Functional constructs*;

Additional Key Words and Phrases: Adequacy, aggregate programming, computational field, core calculus, full abstraction, spatial computing, type inference system, type soundness

ACM Reference format:

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 (January 2019), 55 pages.

<https://doi.org/10.1145/3285956>

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

This work is partially supported by project HyVar (www.hyvar-project.eu, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No.: 644298), by ICT COST Action IC1402 ARVI (www.cost-arvi.eu), and by Ateneo/CSP project RunVar.

Authors' addresses: G. Audrito and F. Damiani, Dipartimento di Informatica, University of Turin, Corso Svizzera 185, Turin 10149, Italy; emails: giorgio.audrito@gmail.com, ferruccio.damiani@unito.it; M. Viroli and D. Pianini, Dipartimento di Informatica – Scienza e Ingegneria, Università di Bologna, Via dell'Università 50, Cesena, 47522, Italy; emails: mirko.viroli@unibo.it, danilo.pianini@unibo.it; J. Beal, Raytheon BBN Technologies, 10 Moulton Street, Cambridge, MA, 02138, USA; email: jakebeal@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org

© 2019 Association for Computing Machinery.

1529-3785/2019/01-ART5 \$15.00

<https://doi.org/10.1145/3285956>

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, and so on. 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 [9] is a paradigm aiming to address this problem by means of the notion of *computational field* [39] (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 processing them to produce output fields to actuators.

The *field calculus* [1, 23, 24] is a tiny functional language providing basic constructs to work with fields. The operational semantics of field calculus 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 [8, 54]. Recent works have also adopted this field calculus as a *lingua franca* to investigate formal properties of resiliency to environment changes [46, 53] and to device distribution [11].

In this article, we present a higher-order extension of the work in Reference [23] to include embedded first-class functions, with the primary goal of allowing field computations to handle functions just like any other value but also making syntax and semantics of the field-based computational model clearer and more coherent. We introduce syntax, typing, denotational semantics, operational semantics, and properties of the higher-order 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 runtime during system operation, spread around the network, and be executed by all (or some) devices, which then coordinate on the collective computation of the new service specified by the injected function.

A key insight and technical result of this article is that the notoriously difficult problem of reconciling local and global behaviour in a complex adaptive system [8, 54] 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 article via *computational adequacy* and a form of *abstraction* (cf. References [20, 52]) that we call *computational abstraction*, 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 article is organised as follows. Section 2 introduces the computational model, syntax, informal semantics, and monomorphic typing of the proposed calculus. Denotational and operational semantics of the calculus are given in Section 3 and Section 4, respectively—notably, the two sections may be read in either order. Section 5 refines the monomorphic typing given in Section 2 and introduces a variant of the Hindley-Milner type system that corresponds to the refined typing. Section 6 uses the refined typing to prove properties of the two semantics, including computational adequacy and abstraction. Finally, Section 7 reviews related work and Section 8 concludes and discusses future directions.

Appendix A gives the proofs of the main results, and Appendix B gives a pervasive computing example.

This article is an extended version of the work in Reference [24], adding a reduced (yet more expressive) and reworked set of constructs, the observation that domain restriction (the *if*-construct in Reference [24]) can be encoded by means of an aggregate function call, a type system, denotational semantics, and computational adequacy and abstraction results.

2 THE HIGHER-ORDER FIELD CALCULUS

We begin by presenting the essential elements of the proposed calculus, the *higher-order field calculus* (HFC), as extended and refined from Reference [24],¹ a tiny functional calculus capturing the essential elements of field computations, much as λ -calculus [17] captures the essence of functional computation and FJ [34] the essence of class-based object-oriented programming. Given the key importance of higher-order features, especially in the toolchain under construction [50], in the following we sometimes refer to this calculus as simply the *field calculus*, especially when there is no confusion with the work in Reference [23], which did not include higher-order features and which this work is intended to supersede.

2.1 Computational Model

To better understand the formal underpinnings to come, we begin by discussing the assumed computational model, in which a given program P is executed by a network of devices, the network being defined by a dynamic neighboring relation that represents physical or logical proximity. The defining property of fields is that they allow us to consider a computation from two different viewpoints: local and global. From the standard “local” viewpoint, a computation is seen as occurring in single devices on a round-based scheme, with a fair and unsynchronized scheduling of device computation rounds across the network. In each round, a device:

- (1) sleeps for some time until it wakes up;
- (2) gathers information about messages received from neighbours while sleeping, in the form of *neighbouring fields* ϕ mapping neighbour device identifiers δ (i.e., unique numbers in a denumerable set D) to values v ;
- (3) perceives contextual information (e.g., through sensors);
- (4) retrieves information that has been stored in the local memory of the device at the end of the previous round;
- (5) performs an evaluation of the program P , hence manipulating the data values received from neighbours, perceived from the context, or retrieved from the local memory;

¹The version of the HFC presented in this article is a minor refinement of the version of HFC presented in Reference [24]. The new version adopts a different syntax (in Reference [24] a Lisp-like syntax was used) and is parametric in the set of the modeled data values (in Reference [24] Booleans, numbers, and pairs were explicitly modeled).

- (6) finally, stores some data value (to be used in the next round) in the local store, emits a message to all neighbours with information about the outcome of computation to enable coordination, produces a value as output (that might be used, e.g., to feed actuators); and
- (7) goes back to sleep.

In further discussion in this article, we say “device δ fires” to mean that device δ performs the steps (2)–(6) at a particular round.

From the “aggregate” (or “global”) viewpoint [56], computation is seen as occurring on the overall network of interconnected devices, interpreted as a single spatial computing machine. The data abstraction manipulated is hence a whole distributed space-time *field evolution* Φ , a data structure mapping computation events ϵ (i.e., points in space-time where and when a device evaluates its program) to associated data values.² Field computations then take field evolutions as input (e.g., from sensors) and produce new field evolutions as outputs (e.g., to feed actuators).

For example, the input of a computation might be a field evolution 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 50m from some device where temperature was greater than 25°C for the last 60s.

A snapshot of the state of the network at a particular time is modeled by the concept of a *computational field*, mapping device identifiers to values. Computational fields may, of course, change over time (e.g., as inputs change or the computation progresses) and in general are not sufficient to fully define the behaviour of an aggregate program, as it may depend on the history of the computational field and its corresponding computation events (i.e., the whole field evolution structure). However, for the relevant class of *self-stabilising* aggregate programs P , the behaviour on field evolutions can be “induced” (at the limit) by a spatial-only behaviour on computational fields [22, 53]. In those cases the computational field abstraction turns out to be quite valuable, allowing one to ignore the time-related fine event structure. In this article, we will not explore how this abstraction can be used to define limit behaviour of programs, but we will sometimes use it for presentation purposes, when investigating the behaviour of time-independent constructs.

Remark 1 (On field-like notions). Notice that the three related notions of *computational field*, *neighbouring field*, and *field evolution* are all distinct. A *computational field* ψ is a mapping from devices to values, that (at a given time) maps each device in the network to the value produced by its most recent firing. A *neighbouring field* ϕ is a situated value stored in a single device summarising received messages (at a given time), and a collection of such fields may be viewed in the aggregate as a “computational field of neighbouring fields” across the network. A *field evolution* Φ , by contrast, is a space-time object with single computation events ϵ (corresponding to device fires) in its domain (instead of devices δ).

2.2 Syntax

Based on the above computational model, the field calculus adopts a functional approach by which overall system behaviour can be declaratively specified and composed in terms of computations resulting in field evolutions.

²Note that this viewpoint can embrace both discrete domains (e.g., networks of devices) and continuous domains (e.g., the environment in which a computation acts); in this article, 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 References [11, 12].

$P ::= \bar{F} e$	program
$F ::= \text{def } d(\bar{x}) \{e\}$	function declaration
$e ::= x \mid (\bar{x}) \stackrel{\tau}{\Rightarrow} e \mid v \mid \text{if } (e) \{e\} \text{ else } \{e\} \mid e(\bar{e})$ $\quad \mid \text{rep}(e)\{(x) \Rightarrow e\} \mid \text{nbr}\{e\}$	expression
$v ::= \phi \mid \ell$	value
$\phi ::= \bar{\delta} \mapsto \bar{\ell}$	neighbouring field value
$\ell ::= f \mid c(\bar{\ell})$	local value
$f ::= b \mid d \mid (\bar{x}) \stackrel{\tau}{\Rightarrow} e$	function value

Fig. 1. Syntax for higher-order field calculus.

Figure 1 presents the syntax of the proposed higher-order field calculus. Following Reference [34], the overbar notation denotes metavariables over sequences and the empty sequence is denoted by \bullet : For example, for expressions, we let \bar{e} range over sequences of expressions, written e_1, e_2, \dots, 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 \bar{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 and *always model an entire field evolution*: 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_e and collect the values obtained in all events in that domain D_e and by doing so form a field evolution over D_e , representing a time-varying distributed structure.

Syntactically, an expression can be a *variable* x , used as function formal parameter; an *anonymous function expression* $(\bar{x}) \stackrel{\tau}{\Rightarrow} e$; a *value* v (possibly of functional type); a *branching expression* $\text{if } (e) \{e\} \text{ else } \{e\}$; a *function call* $e(\bar{e})$; a *rep-expression* $\text{rep}(e_0)\{(x) \Rightarrow e_1\}$ modelling time evolution; or an *nbr-expression* $\text{nbr}\{e\}$ modelling device-to-neighbourhood interaction. Let the set of free variables in an expression e be denoted by $\text{FV}(e)$,³ say, that an expression e is *closed* if $\text{FV}(e) = \bullet$, and assume the main expression of any program must be closed. In Section 2.3, we informally describe the meaning of these constructs, while the full formal treatment of denotational and operational semantics will be given in the remainder of the article.

An anonymous function is an expression $(\bar{x}) \stackrel{\tau}{\Rightarrow} e$, where \bar{x} are the formal parameters, e is the body, and τ is a *tag*. Tags τ do not appear in source programs: When the evaluation starts each anonymous function expression $(\bar{x}) \Rightarrow e$ occurring in the program is converted into a tagged anonymous function expression by giving it a tag that is uniquely determined by its syntactical representation, i.e., $e_1 = (\bar{x}') \Rightarrow e'$ and $e_2 = (\bar{x}'') \Rightarrow e''$ get the same tag if and only if they are syntactically equal (modulo renaming of bound variables).⁴ As we will see in Section 2.3, tags are used to define the semantics of function call. In the remainder of this article, we will use the phrase *name of a function* to refer both to the name of a user-defined or built-in function and to the tag of an anonymous function.

³This set is defined as usual, given that the only binding construct is $(x) \Rightarrow e$ (occurring both alone or inside a rep), which binds occurrences of x in e .

⁴If e_1 and e_2 contain free variables, then being syntactically equal implies that each of these variables is bound both in e_1 and e_2 by the same binder in the surrounding context. Recall that the whole program do not contain free variables, so the variables that are free in e_1 and e_2 are bound in the whole program—and the tags are given considering the whole program.

A value can be either a *neighbouring field value* ϕ or a *local value* ℓ . At a given event where a device δ fires, a local value represents data produced by δ , whereas a neighbouring field value is a map associating local values ℓ to neighbours 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 the local environment, e.g., to estimate distances to neighbours). As such, neighbouring field values *are not allowed to appear in source programs but only be computed dynamically during computation rounds*. On the contrary, local values ℓ can be denoted to represent a constant-valued field evolution that maps each event to that value (e.g., value 1 represents a field evolution mapping each event to 1).

Given that the calculus is higher-order, a local value can be

- either a *data value* $c(\ell_1, \dots, \ell_m)$, consisting of a data-constructor c of arity $m \geq 0$ applied to m local value arguments ℓ_1, \dots, ℓ_m —data values, simply written c when $m = 0$, can be Booleans `True` and `False`, numbers, strings, or structured values like pairs (e.g., `Pair(3, Pair(False, 5))`) or lists (e.g., `Cons(2, Cons(4, Nil))`);
- or a *function value* f , consisting of either a built-in function name b , a declared function name d , or a closed anonymous function.

It should be noted that for the purpose of defining a foundational calculus, data-expressions and built-in functions representing purely functional operators could be dropped: however, we have decided to include them since they simplify using the calculus to formalise non-trivial examples. Furthermore, built-in functions on neighbouring field values are crucial in determining the expressiveness of the language: If no such operators are given, then no effective communication between devices is possible. As a reasonably complete basis, in the remainder of this article we assume that the following operators (reminiscent of Reference Dean and Ghemawat [27]) are available:

- `map-hood`(f, ϕ_1, \dots, ϕ_n), which applies the n -ary ($n \geq 0$) function f pointwise to given fields ϕ_1, \dots, ϕ_n , returning a new field as output—note that this requires that ϕ_1, \dots, ϕ_n must all have the same domain; and
- `fold-hood`(f, ϕ), which reduces a neighbouring field ϕ to a single value by repeatedly applying function f over the multi-set of values mapped to by devices in ϕ .

More details on built-in functions are given in Section 2.4.

2.3 Informal Semantics

While values trivially result in a field that is constant in both space and time, the other four kinds of expression (`nbr`-expressions, `rep`-expressions, branching, and function calls) represent the core field manipulation constructs provided by the calculus. The value produced by evaluating such an expression at each event may depend on both the particular device that is evaluating it, and also on the last events of its neighbours (i.e., the events whose output messages reached the current device in its previous sleeping phase).

- (1) *Time evolution*: `rep`(e_0){ $(x) \Rightarrow e$ } is a “repeat” construct for dynamically changing fields—note that, since the anonymous function $(x) \Rightarrow e$ is part of the syntax of the `rep`-expression, it is not given a tag. At each asynchronous computation round each device δ yields, as value of the `rep`-expression, the result of the application of the anonymous function $(x) \Rightarrow e$ to the value of the `rep`-expression at the previous round—whenever the `rep`-expression has not been evaluated in the previous round (e.g., in the first round, or when the `rep`-expression occurs in a branch of an `if`-expression that was not taken in the previous round), then the anonymous function $(x) \Rightarrow e$ is applied to the value

of the *initialization expression* e_0 . For instance, $\text{rep}(0)\{(x) \Rightarrow +(x, 1)\}$ counts how many rounds each device has computed.

- (2) *Neighbouring field construction*: $\text{nbr}\{e\}$ models device-to-neighbour interaction by representing a field evolution of neighbouring field values: Each event is associated to a value ϕ , which in turn maps neighbour devices δ to their most recent available value of e (e.g., obtained via periodic broadcast, as discussed in Section 2.1). Such neighbouring field values can then be manipulated and summarised with built-in functions. For instance, $\text{min-hood}(\text{nbr}\{e\})$ maps each device to the minimum value of e across its neighbourhood.
- (3) *Domain restriction*: $\text{if } (e_0) \{e_1\} \text{ else } \{e_2\}$ is a lazy-evaluating branch construct, computing e_1 on the devices in the restricted domain D_{True} where e_0 is true, and e_2 on the devices in its complement D_{False} , similar to branches in most common programming languages. However, in field calculus branching has peculiar consequences: As a device δ in D_{False} does not compute e_1 , it also
 - does not evaluate any nbr -expression e' contained in e_1 , making impossible for δ 's neighbours in D_{True} to obtain δ 's value for e' ; conversely, nbr -expressions e'' contained in e_2 are not shared from D_{True} to D_{False} ; and
 - if δ evaluated e_1 instead of e_2 in its previous round, all rep -expressions in e_2 start from scratch (i.e., are evaluated by considering the values of their initialization expressions); similarly, the values stored for rep -expressions in e_1 are lost so that, in the first subsequent round in which e_1 will be evaluated on δ , those rep -expressions will also start from scratch.

It follows that the evaluation of e_1 and e_2 proceeds in the corresponding subdomains D_{True} or D_{False} in *isolation*, without any possible communication between the two branches.

This process, which we call *clustering*, is a necessary consequence of a lazy branching construct in a compositional language with implicit message-passing through value sharing (e.g., nbr -expressions). However, it is also a crucial *feature* of a spatial language, as it allows one to easily and efficiently restrict computations to subdomains in a declarative way, an important building block for many practical application scenarios.

- (4) *Function call*: $e(e_1, \dots, e_n)$ models a function call, where $n \geq 0$ and e evaluates to a field of function values. If the field is not constant, then the application is evaluated separately in each *cluster* (subdomain of events where e evaluates to function values with the same name⁵), acting as a multi-way branch similar to how branching is described for domain restriction. As before, this feature is necessary as values of nbr -expressions in functions different than the current one cannot be evaluated and hence are not available to be shared. 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 function b , then $e(e_1, \dots, e_n)$ maps each device to the result of applying b to the values at the same device of its $n \geq 0$ arguments e_1, \dots, 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 computational field 3, also written $1 + 2$ for readability as for any other binary built-in function. Alternatively, b can be an *environment-dependent operator*, modelling a sensor—for instance, 0-ary sns-temp returns the local value of temperature in each device δ where the built-in function call is evaluated, and the 0-ary nbr-range operator returns (in each device δ where the built-in function call is

⁵Recall that the *name* of a function can be either the name of a declared/built-in function or the tag of an anonymous function (see the explanation in Section 2.2).

evaluated) a neighbouring field value mapping neighbours of δ to estimates of their current distance from δ .

- If f is not a built-in function, then it can be a declared function d with corresponding declaration $\text{def } d(x_1, \dots, x_n) \{e\}$, or an anonymous function value $(x_1, \dots, x_n) \xRightarrow{\tau} e$; then, expression $e(e_1, \dots, e_n)$ maps an event to the result of evaluating the closed expression e_0 obtained from the body e of the function f by replacing the occurrences of the formal parameters x_1, \dots, x_n with the values of the expressions e_1, \dots, e_n .

Remark 2 (On function equality). The semantic of function call given above considers two function values f_1 and f_2 as “equal” when they have the same name. This design choice is carried over in the meaning of the built-in operator $=$ when applied to function values f_1 and f_2 , so that $=(f_1, f_2)$ holds precisely when f_1 and f_2 have the same name.⁶ For built-in functions and declared functions this notion of equality (provided by the built-in operator $=$) coincides with syntactical equality (considering names as part of the syntax). This is not true for anonymous function values: although different occurrences of the same anonymous function expression $(x) \xRightarrow{\tau} e$ in a source program get the same tag (see Section 2.2), syntactically equal anonymous functions containing a *non-closed anonymous function subexpression* may evaluate to syntactically different anonymous function values that have the same tag (and are therefore considered as equal by the built-in operator $=$). For example, any occurrence of the anonymous function expression $e' = (x_1) \xRightarrow{\tau_1} (x_2) \xRightarrow{\tau_2} \text{min-hood}(\text{nbr}\{x_1\})$ in a program yields the anonymous function value $f = (x_1) \xRightarrow{\tau_1} (x_2) \xRightarrow{\tau_2} \text{min-hood}(\text{nbr}\{x_1\})$ that, when applied to the two different values 3 and 4, returns two syntactically different anonymous function values:

- $f_1 = f(3) = (x_2) \xRightarrow{\tau_2} \text{min-hood}(\text{nbr}\{3\})$
- $f_2 = f(4) = (x_2) \xRightarrow{\tau_2} \text{min-hood}(\text{nbr}\{4\})$

such that $=(f_1, f_2)$ holds. It is worth observing that this semantics of function call ensures that curried functions behave as their non-curried counterparts. For example the following four programs:

- $P_1 = e'(\text{sns-temp})(\text{True})$
- $P_2 = ((x_1, x_2) \xRightarrow{\tau} \text{min-hood}(\text{nbr}\{x_1\}))(\text{sns-temp}, \text{True})$
- $P_3 = \text{def } d(x_1, x_2) \{\text{min-hood}(\text{nbr}\{x_1\})\} \ d(\text{sns-temp}, \text{True})$
- $P_4 = \text{def } d(x_1) \{(x_2) \xRightarrow{\tau} \text{min-hood}(\text{nbr}\{x_1\})\} \ d(\text{sns-temp})(\text{True})$

all behave in the same way (they map each device to the minimum temperature in its neighbourhood). We remark that disallowing the equality operator on functions, or changing the definition of function call and functional equality with stricter notions (for instance, assigning a different tag to each occurrence of an anonymous function expression in the source program), would preserve the properties of the calculus and, in particular, the correspondence between its operational and denotational semantics, but reserve exploration of these options for future work.

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”;

⁶In ML-like languages [42], there is no equality operator for function values. However, since the alignment process of the field calculus induces a notion of equality for function values, we decided to adopt it as the semantics of the built-in operator $=$ on function values.

(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 3 (On if-expressions). Any if-expression $e = \text{if } (e_0) \{e_1\} \text{ else } \{e_2\}$ can be equivalently rewritten through a function call. If e_1 and e_2 are not syntactically equal, then e can be rewritten as:

$$\text{mux}(e_0, () \Rightarrow e_1, () \Rightarrow e_2)(),$$

where function `mux` is a built-in function multiplexer that takes three arguments (a Boolean and two other arguments that must be of the same type) and returns the second argument if the first argument is `True` and the third argument otherwise. If e_1 and e_2 are syntactically equal, then we need to use the following more general rewriting to ensure that the two function calls are kept distinct:

$$\text{mux}(e_0, () \Rightarrow \text{snd}(\text{pair}(\text{True}, e_1)), () \Rightarrow \text{snd}(\text{pair}(\text{False}, e_2)))(),$$

where the built-in function `pair` builds a `pair`⁷ and the built-in function `snd` extracts the second component of a pair. In the remainder of this article, we will thus omit to give the typing, operational and denotational semantics of `if` statements, and instead consider them as syntactic sugar for the expression above. We retain the `if` construct in our syntax, however, for its familiarity, convenience, and intuitive value in examples.

Remark 4 (On termination). As our syntax allows recursive functions, termination of a device firing is clearly not decidable. In the rest of the article, we assume without loss of generality for the results of this article that a decidable subset of the termination fragment is considered. Such a fragment could be identified, e.g., by means of some static analysis technique for termination (see, e.g., Reference [30]).

2.4 A Monomorphic Type System for HFC

In this section, we introduce a type system for assigning a monomorphic type to the main expression e of an HFC program $P = \bar{F} e$. This type system will be used (in Section 3) to ground the denotational semantics of the calculus. We first consider the typing rules for expressions e (in Section 2.4.1) and then the typing rules for user-defined functions and programs (in Section 2.4.2).

2.4.1 Monomorphic Typing for Expressions. The monomorphic version of the type system TA_λ for the lambda calculus [33] can be straightforwardly extended to cover the peculiar constructs of field calculus.

The syntax of types is given in Figure 2 (top). A *type* T is either a local type or a field type. A *local type* L is either a built-in type B (numbers, Booleans, pairs, lists etc.), or the type of a function $(\bar{T}) \rightarrow T$ (possibly of arity zero). Note that a function always has local type, regardless of the local or field type of its arguments. A *field type* F is the type $\text{field}(L)$ of a neighbouring field whose range contains values of local type L . Given $\bar{L} = L_1, \dots, L_n$ ($n \geq 0$), we write $\text{field}(\bar{L})$ as short for $\text{field}(L_1), \dots, \text{field}(L_n)$.

⁷We have used the built-in function `pair` instead of the data-constructor `Pair` since e_1 and e_2 might not be values, and data-constructors must be applied to values.

Monomorphic types:	
$T ::= L \mid F$	type
$L ::= B \mid (\bar{T}) \rightarrow T$	local type
$F ::= \text{field}(L)$	field type
Expression monomorphic typing:	
$\boxed{S; \mathcal{M} \vdash e : T}$	
$\frac{[M\text{-VAR}]}{S; \mathcal{M}, x : T \vdash x : T}$	$\frac{[M\text{-DAT}]}{(\bar{L}) \rightarrow L \in S_0(c) \quad S_0; \emptyset \vdash \bar{e} : \bar{L}} \quad S; \mathcal{M} \vdash c(\bar{e}) : L$
$\frac{[M\text{-FLD}]}{S; \mathcal{M} \vdash \phi : \text{field}(L)} \quad \phi = \bar{\delta} \mapsto \bar{e} \quad S; \mathcal{M} \vdash \bar{e} : L$	$\frac{[M\text{-N-FUN}]}{(\bar{T}) \rightarrow T \in S(g)} \quad S; \mathcal{M} \vdash g : (\bar{T}) \rightarrow T$
$\frac{[M\text{-A-FUN}]}{S; \mathcal{M}, \bar{x} : \bar{T} \vdash e : T} \quad S; \mathcal{M} \vdash \phi : \text{field}(L)$	$\frac{[M\text{-APP}]}{S; \mathcal{M} \vdash e : (\bar{T}) \rightarrow T \quad S; \mathcal{M} \vdash \bar{e} : \bar{T}} \quad S; \mathcal{M} \vdash \phi : \text{field}(L)$
$\frac{[M\text{-REP}]}{S; \mathcal{M} \vdash \text{rep}(e_1)\{x\} \Rightarrow e_2 : T} \quad S; \mathcal{M}, x : T \vdash e_2 : T$	$\frac{[M\text{-NBR}]}{S; \mathcal{M} \vdash \text{nbr}\{e\} : \text{field}(L)} \quad S; \mathcal{M} \vdash e : L$

Fig. 2. Monomorphic typing for HFC expressions.

Type environments, ranged over by \mathcal{M} and written $\bar{x} : \bar{T}$, are used to collect type assumptions for program variables (i.e., the formal parameters of functions and the variables introduced by the rep-construct). *Local-type-set environments*, ranged over by S and written $\bar{g} : \bar{L}\text{set}$, are used to associate a suitable non-empty set of function types $L\text{set}$ to each data constructor, built-in function or user-defined function g . In particular, the distinguished *built-in local-type-set environment* S_0 associates to each data constructor c a non-empty set of types $S_0(c)$ of the form $(\bar{L}) \rightarrow L$ (i.e., function types that do not involve field types) and to each built-in function b a non-empty set $S_0(b)$ of function types—Figure 3 shows the set of types for the data constructors and built-in functions used throughout this article.⁸ 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). Notice that the built-in function map-hood accepts a first function argument of type $(L_1, \dots, L_n) \rightarrow L$ ($n \geq 0$), followed by n arguments of type $\text{field}(L_1), \dots, \text{field}(L_n)$, respectively.

The typing rules are given in Figure 2 (bottom). The typing judgement for expressions is of the form “ $S; \mathcal{M} \vdash e : T$ ”, to be read: “ e has type T under the type-set assumptions S (for data constructors, built-in and defined functions) and the type assumptions \mathcal{M} ” (for the program variables occurring in e).

As a standard syntax in type systems [34], given $\bar{T} = T_1, \dots, T_n$ and $\bar{e} = e_1, \dots, e_n$ ($n \geq 0$), we write $S; \mathcal{M} \vdash \bar{e} : \bar{T}$ as short for $S; \mathcal{M} \vdash e_1 : T_1 \cdots S; \mathcal{M} \vdash e_n : T_n$. Note that the type rules are syntax directed, so they straightforwardly describe a type inference algorithm, and that there is no need for a rule typing neighbouring field values ϕ , since they are not allowed to appear in source programs.

Rule [M-VAR] (for variables) is standard: It looks up the type assumptions for x in \mathcal{M} .

Rule [M-DAT] (for data values) allows us to assign to the data constructor c any type in $S_0(c)$ that meets the types of the given arguments, which need to be (local) *values*—the latter allows us to recognize whether a well-typed expression is a value by only checking its outermost data

⁸Notice that type-sets in S_0 are allowed to be infinite (even though finitely presented), whereas type-sets assigned to user-defined functions are always finite.

Built-in constructors:	
$S_0(\text{True})$	$= \{() \rightarrow \text{bool}\}$
$S_0(\text{False})$	$= \{() \rightarrow \text{bool}\}$
$S_0(0)$	$= \{() \rightarrow \text{num}\}$
$S_0(\text{Pair})$	$= \{(L_1, L_2) \rightarrow \text{pair}(L_1, L_2) \mid L_1, L_2 \text{ local types}\}$
$S_0(\text{Nil})$	$= \{() \rightarrow \text{list}(L) \mid L \text{ local type}\}$
$S_0(\text{Cons})$	$= \{(L, \text{list}(L)) \rightarrow \text{list}(L) \mid L \text{ local type}\}$
Pure built-in functions (independent from the current device and value-tree environment):	
$S_0(\text{fst})$	$= \{(\text{pair}(L_1, L_2)) \rightarrow L_1 \mid L_1, L_2 \text{ local types}\}$
$S_0(\text{snd})$	$= \{(\text{pair}(L_1, L_2)) \rightarrow L_2 \mid L_1, L_2 \text{ local types}\}$
$S_0(\text{pair})$	$= \{(L_1, L_2) \rightarrow \text{pair}(L_1, L_2) \mid L_1, L_2 \text{ local types}\}$
$S_0(\text{head})$	$= \{(\text{list}(L)) \rightarrow L \mid L \text{ local type}\}$
$S_0(\text{tail})$	$= \{(\text{list}(L)) \rightarrow \text{list}(L) \mid L \text{ local type}\}$
$S_0(\text{min-hood})$	$= \{(\text{field}(L)) \rightarrow L \mid L \text{ local type}\}$
$S_0(\text{min-hood+})$	$= \{(\text{field}(L)) \rightarrow L \mid L \text{ local type}\}$
$S_0(\text{pick-hood})$	$= \{(\text{field}(L)) \rightarrow L \mid L \text{ local type}\}$
$S_0(\text{map-hood})$	$= \{(\bar{L} \rightarrow L, \text{field}(\bar{L})) \rightarrow \text{field}(L) \mid \bar{L}, L \text{ local types}\}$
$S_0(\text{fold-hood})$	$= \{((L, L) \rightarrow L, \text{field}(L)) \rightarrow L \mid L \text{ local type}\}$
$S_0(\text{mux})$	$= \{(\text{bool}, L, L) \rightarrow L \mid L \text{ local type}\}$
$S_0(\text{and}), S_0(\text{or})$	$= \{(\text{bool}, \text{bool}) \rightarrow \text{bool}\}$
$S_0(+), S_0(-), S_0(*), S_0(/)$	$= \{(\text{num}, \text{num}) \rightarrow \text{num}\}$
$S_0(=)$	$= \{(T, T) \rightarrow \text{bool} \mid T \text{ type}\}$
Non-pure built-in functions (depend on the current device and value-tree environment):	
$S_0(\text{sns-range})$	$= \{() \rightarrow \text{num}\}$
$S_0(\text{sns-injection-point})$	$= \{() \rightarrow \text{bool}\}$
$S_0(\text{sns-injected-function})$	$= \{() \rightarrow (() \rightarrow \text{num})\}$
$S_0(\text{nbr-range})$	$= \{() \rightarrow \text{field}(\text{num})\}$
$S_0(\text{uid})$	$= \{() \rightarrow \text{num}\}$

Fig. 3. Local type sets for the data constructors and built-in functions used throughout this article.

constructor, which is convenient for later proofs. Notice that different occurrences of c in an expression e may be assigned different types. For convenience of presentation, in the following we assume that every constructor c comes with an associated built-in function b_c that behaves like $(\bar{x}) \Rightarrow c(\bar{x})$ and is such that $S_0(b_c) = S_0(c)$.

Rule [M-FLD] (for field values) assigns type $\text{field}(L)$ to $\phi = \bar{\delta} \mapsto \bar{\ell}$ if each ℓ_i has type L .

Rule [M-N-FUN] (for built-in and user-defined function names) allows assignment of the built-in or user-defined function g to any of the types in $S(g)$. Notice that different occurrences of g in an expression e may be assigned different types.

Rule [M-A-FUN] (for anonymous functions) and Rule [M-APP] (for function applications) are standard.

Rule [M-REP] (for rep-expressions) ensures that both the variable x , its initial value e_1 and the body e_2 have the same type.

Rule [M-NBR] (for nbr-expressions) ensures that the body e of the expression has a local 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).

NOTATION 5 (BUILT-IN FUNCTIONS ON FIELDS). *We use the convention that if b is a built-in unary operator with local argument and return type, $b[f]$ denotes the corresponding operator on neighbouring field values that applies b pointwise to its argument. In other words: $b[f](\phi)$, which is equivalent to $\text{map-hood}(b, \phi)$, 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, then a notation such as $b[f, 1]$ or $b[1, f, f]$ is used to specify which parameters have to be promoted to neighbouring field values. For instance, at each device, $+ [f, f](\phi_1, \phi_2)$ gives a neighbouring field value mapping a neighbour δ to the sum of values of ϕ_1 and ϕ_2 at δ . Typing*

of such extended operators is defined accordingly, e.g.,

$$\begin{aligned} \mathcal{S}_0(\text{pair}[l, f]) &= \{(L_1, \text{field}(L_2)) \rightarrow \text{field}(\text{pair}(L_1, L_2)) \mid L_1, L_2 \text{ local types}\} \\ \mathcal{S}_0(\text{pair}[f, f]) &= \{(\text{field}(L_1), \text{field}(L_2)) \rightarrow \text{field}(\text{pair}(L_1, L_2)) \mid L_1, L_2 \text{ local types}\}. \end{aligned}$$

2.4.2 Monomorphic Typing for User-defined Functions and Programs. The type system in the previous section can be straightforwardly extended to cover defined functions and whole programs by means of the following rules.

<p>[M-FUNCTION]</p> $\frac{\mathcal{S}, d : \{\bar{T}^{(1)} \rightarrow T^{(1)}\}; \bar{x} : \bar{T}^{(1)} \vdash e : T^{(1)} \cdots \mathcal{S}, d : \{\bar{T}^{(m)} \rightarrow T^{(m)}\}; \bar{x} : \bar{T}^{(m)} \vdash e : T^{(m)} \quad m \geq 1}{\mathcal{S} \vdash \text{def } d(\bar{x}) \{e\} : \{\bar{T}^{(1)} \rightarrow T^{(1)}, \dots, \bar{T}^{(m)} \rightarrow T^{(m)}\}}$ <p>[M-PROGRAM]</p> $\frac{\begin{array}{l} F_i = (\text{def } d_i(_) _) \quad \mathcal{S}_{i-1} \vdash F_i : Lset_i \quad \mathcal{S}_i = \mathcal{S}_{i-1}, d_i : Lset_i \quad (i \in 1 \dots n) \\ \mathcal{S}_n; \emptyset \vdash e : T \end{array}}{\mathcal{S}_0 \vdash F_1 \cdots F_n e : T}$

Rule [M-FUNCTION] allows assignment of a user-defined function $\text{def } d(\bar{x}) \{e\}$ to a non-empty set of types. Rule [M-PROGRAM] allows assignment of a non empty-set of types to each of the user-defined functions, and then assigns to the program the type of its main expression.

Remark 6 (On typing user-defined functions). To simplify the presentation, we do not consider the issue of typing mutually recursive user-defined functions, which can be addressed by exploiting standard techniques (see, e.g., Reference [42]). In Section 5, we will present a variant of the Hindley-Milner type system [21] that types a program $P = F_1 \cdots F_n e$ ($n \geq 0$) by first assigning a type scheme to each used defined function F_i ($1 \leq i \leq n$), and then using these type schemes to assign a type to each occurrence of a name of the user-defined function in the main expression e . Furthermore, this type system will enforce some additional restrictions that (in Section 6) will be used to prove properties of well-typed programs—as we will show in Section 5, the type system presented in this section allows assignment to the main expression e of a program of all the monomorphic types that can be assigned to it by the type system of Section 5.

Example 2.1. As an illustration of the syntax, semantics, and typing just discussed, consider the following simple program:

```
def counter() { rep (0) {(x) => x+1} } ;; has type: () -> num
def lowest-nbr(fun) { min-hood(nbr{ fun() }) } ;; has types: (() -> L) -> L
if (counter() < 10) { 0 } else { lowest-nbr(uid) } ;; has type: num
```

The first line defines a function `counter`, which uses `rep` to count the number of rounds that the function has been executed at each device (see Section 2.3, discussion on *time evolution*). The second defines a higher-order function `lowest-nbr`, which takes a function `fun` as input, evaluates it, shares the results of evaluation with each device's neighbourhood (see *neighbouring field construction*), and returns the lowest neighbouring value. Finally, the main expression makes use of these user-defined functions in a branching statement (see *domain restriction*), where any device that has executed less than 10 times returns zero, while all others call `lowest-nbr` on `uid` built-in function, returning the lowest unique identifier of any device that has executed at least 10 times in their neighbourhood. Applying the monomorphic type system, we may identify this program as one that can be consistently typed, with the function `counter` having type $() \rightarrow \text{num}$, the function

lowest-nbr having type $((\rightarrow \text{num}) \rightarrow \text{num})$ (actually, it has any type of the form $((\rightarrow L) \rightarrow L)$, where L is a local type), and the overall program having a type of num .

3 DENOTATIONAL SEMANTICS

In this section, we introduce a denotational semantics for the field calculus. In designing 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 [40, 47, 60]): 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., References [5, 25]), 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.

To reflect these characteristics of field calculus, the denotational semantics of an expression is given in terms of the resulting field evolution (see Remark 1). That is, a 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 neighbouring fields and formulating functions as mathematical operators on field evolutions 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”, the field evolution, 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 4) is instead more useful in designing a platform for the equivalent distributed execution of field computations on actual devices.

3.1 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 assume that each ϵ incorporates all the relevant information about the corresponding event, e.g., the involved device and its sensor state at the time the event happened.

We model the neighbour relationship as a global-level, fixed predicate $\text{neigh}(\epsilon, \epsilon')$ that 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 $\text{neigh}(\epsilon, \epsilon')$ satisfies the following properties:

- (1) the graph on E induced by neigh is a well-founded DAG (i.e., a directed acyclic graph such that each node can be reached by a finite set of nodes,⁹ called the *backward star* of the node);

⁹This prohibits computations that extends indefinitely in the past (without a start), but does not rule out the possibility of computations over an infinite amount of time in the future or involving an infinite number of devices. Even though such computations are not physically realisable, they are sometimes convenient to model limit behaviour of discrete systems.

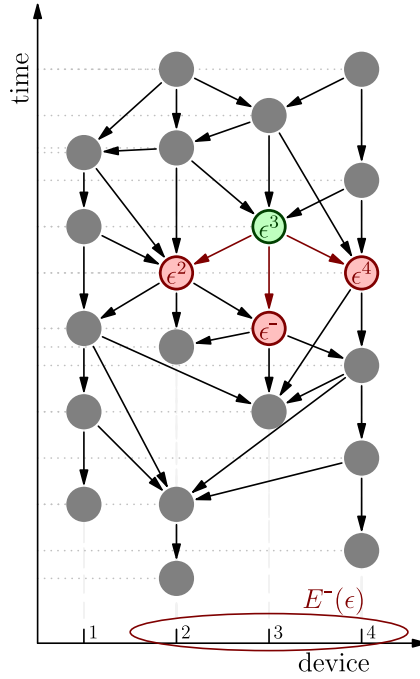


Fig. 4. A sample neighbourhood graph on four asynchronously firing devices. The neighbours of the green event $\epsilon = \epsilon^3$ are shown in red, forming a neighbourhood over devices 2, 3, and 4.

- (2) every ϵ' linked from ϵ has a different label $\delta_{\epsilon'}$, that is, there exists no ϵ', ϵ'' such that $\epsilon' \neq \epsilon'', \delta_{\epsilon'} = \delta_{\epsilon''}$ and both $neigh(\epsilon, \epsilon')$ and $neigh(\epsilon, \epsilon'')$ hold; and
- (3) every ϵ' is linked from at most one ϵ with the same label $\delta_{\epsilon} = \delta_{\epsilon'}$, that is, there exists no ϵ_0, ϵ_1 such that $\delta_{\epsilon_0} = \delta_{\epsilon_1} = \delta_{\epsilon'}$ and both $neigh(\epsilon_0, \epsilon')$ and $neigh(\epsilon_1, \epsilon')$ 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 4 shows a sample neighbourhood graph involving four devices, each firing from four to six times. Notice that device 2 is rebooted 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 4. 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 $\delta = \delta_{\epsilon^\delta}$ and $neigh(\epsilon, \epsilon^\delta)$ if $\delta \neq \delta_\epsilon$, or $\epsilon^\delta = \epsilon$ in case $\delta = \delta_\epsilon$. 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 Figure 4 for event $\epsilon = \epsilon^3$. We also define $E^-(\epsilon)$ 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, $E^-(\epsilon) = \{2, 3, 4\}$ for the green event $\epsilon = \epsilon^3$ in the picture above.

We chose such a generic approach to model neighbouring 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 3.1 (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 $\text{neigh}(\epsilon, \epsilon')$ could be defined from a set of *paths* \mathbf{P} for moving devices, labels t_ϵ modelling passage of time, a timeout value t_d , and a predicate $\text{neigh}(p, p')$ between positions, where

- $\text{neigh}(p, p')$ is a global-level, fixed reflexive and symmetric predicate that 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}^+ (domain) to the set of possible positions (codomain), defined on the union of a finite number of disjoint closed intervals (the time intervals in which the device is turned on).¹⁰
- $t_d \in \mathbb{R}^+$ models a timeout expiration after which non-communicating devices are considered “removed,” allowing adaptation of the network to device removal and topology changes.
- $\text{neigh}(\epsilon, \epsilon')$ holds if and only if
 - (1) $t_{\epsilon'} \in [t_\epsilon - t_d, t_\epsilon]$ (i.e., ϵ' happened in the time interval of size t_d before ϵ);
 - (2) $\mathbf{P}(\delta_\epsilon)$ is defined in the interval $[t_{\epsilon'}, t_\epsilon]$ (i.e., δ_ϵ was constantly turned on during the time between events ϵ' and ϵ);
 - (3) $\text{neigh}(\mathbf{P}(\delta_\epsilon)(t_{\epsilon'}), \mathbf{P}(\delta_{\epsilon'})(t_{\epsilon'}))$ holds (i.e., the two devices were neighbours when ϵ' happened);
 - (4) there exists no further event ϵ'' with $\delta_{\epsilon''} = \delta_{\epsilon'}$ and $t_{\epsilon''} > t_{\epsilon'}$ satisfying the above conditions (i.e., ϵ' is the last firing of $\delta_{\epsilon'}$ recorded by δ_ϵ before ϵ).

3.2 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. Following a standard approach, the denotation of a type consists of a set S such that denotations of expressions of that type are taken from S —denotation of expressions will be presented in next section. The denotational semantics of a type is given by two intertwined functions: a function $\mathcal{V}[\![\cdot]\!]$ mapping a type T to a set of local value denotations (i.e., values at individual devices), and a function $\mathcal{T}[\![\cdot]\!]$ 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, then we assume that $\mathcal{V}[\![B]\!]$ is given. For derived types, $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{T}[\![\cdot]\!]$ are altogether defined by rules:

$$\begin{aligned} \mathcal{T}[\![T]\!] &= \mathbf{E} \rightarrow \mathcal{V}[\![T]\!] \\ \mathcal{V}[\![\text{field}(L)]\!] &= \mathbf{D} \rightarrow \mathcal{V}[\![L]\!] \\ \mathcal{V}[\![(T_1, \dots, T_n) \rightarrow T]\!] &= \mathbf{F} \times (\mathcal{T}[\![T_1]\!] \times \dots \times \mathcal{T}[\![T_n]\!] \rightarrow \mathcal{T}[\![T]\!]), \end{aligned}$$

where $\mathbf{E} \rightarrow \mathcal{V}[\![T]\!]$ (respectively, $\mathbf{D} \rightarrow \mathcal{V}[\![T]\!]$) is the set of partial functions from \mathbf{E} (respectively, \mathbf{D}) to $\mathcal{V}[\![T]\!]$, \mathbf{F} is a set of function tags uniquely characterizing each function, and $\mathcal{T}[\![T_1]\!] \times \dots \times \mathcal{T}[\![T_n]\!] \rightarrow \mathcal{T}[\![T]\!]$ ($n \geq 0$) is the set of all partial functions f from $\mathcal{T}[\![T_1]\!] \times \dots \times \mathcal{T}[\![T_n]\!]$ to $\mathcal{T}[\![T]\!]$ that are defined for $\bar{\Phi} = \Phi_1 \times \dots \times \Phi_n$ if and only if $\text{dom}(\Phi_1) = \dots = \text{dom}(\Phi_n)$ (the domain of computation is coherent among all arguments), and in such case $\text{dom}(f(\bar{\Phi})) = \text{dom}(\bar{\Phi})$ (the domain of computation is not changed by function application). We use the notation $\text{dom}(\bar{\Phi})$ to denote $\text{dom}(\Phi_i)$ for any Φ_i in $\bar{\Phi}$, whenever these domains are assumed to be equal.

¹⁰We 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.

Remark 7 (Denotational semantics of 0-ary functions). In the denotational semantics of function types, we assumed that the domain of computation can be extracted from the function arguments as $\text{dom}(\bar{\Phi})$. However, this is not the case if we allow for 0-ary functions, for which $\text{dom}(\bar{\Phi})$ is undefined. This issue can be solved by adding a first argument of unit type to each function, which is thus provided in every corresponding function call. For sake of readability, in the presentation of the semantics, this additional argument is left implicit.

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

The local value denotation of a field type $\text{field}(L)$ 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 D) to local value denotations of the corresponding local type.

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

- The function tag in F (e.g., a syntactic function value as in Figure 1), needed to reflect the choice to compare functions by *syntactic* equality instead of *semantic* equality, which would not allow a computable operational semantics (see Remark 2). In fact, the presence of such tags is used to grant that two differently specified but identically behaving functions f, f' get distinct denotations.
- A mapping from input field evolutions in $\mathcal{T}[[T_1]], \dots, \mathcal{T}[[T_n]]$ to an output field evolution in $\mathcal{T}[[T]]$.

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` constructs 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 E . In particular, this fact implies that a field evolution Φ of function type and domain E is built of functions $\text{snd}(\Phi(\epsilon))$ that can take arguments of arbitrary domain, *including domains* $E' \not\subseteq E$. Notice that this property grants that a local denotational function value can be meaningfully moved around devices (through constructs `nbr`, `rep`).

Notice that the definition of $\mathcal{V}[(T_1, \dots, T_n) \rightarrow T]$ 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 (on given subdomains): this reflects the assumption that every function call is guaranteed to terminate (see Remark 4).

3.3 Denotational Semantics of Expressions

The denotational semantics of a well-typed expression e of type T in domain E under assumptions $X = \bar{x} \mapsto \bar{\Phi}$ is written $\mathcal{E}[[e]]_X^E$ and yields a field evolution in $\mathcal{T}[[T]]$ 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 $C[[c]]$ in $\mathcal{V}[[L_1]] \times \dots \times \mathcal{V}[[L_n]] \rightarrow \mathcal{V}[[L]]$ ($n \geq 0$) translating the behaviour of built-in constructors c of type $(L_1, \dots, L_n) \rightarrow L$ ¹¹ and by the function $\mathcal{B}[[b]]$ in $\mathcal{T}[[T_1]] \times \dots \times \mathcal{T}[[T_n]] \rightarrow \mathcal{T}[[T]]$ ($n \geq 0$) translating the behaviour of built-in

¹¹Since a constructor does not depend on the environment, we do not need an element of $\mathcal{V}[(\bar{L}) \rightarrow L]$ in this case.

functions b of type $(T_1, \dots, T_n) \rightarrow T$ to denotational values and possibly depending on sensor values and global environment status.¹² Examples of $\mathcal{B}[\![b]\!]$ are given in Section 3.4.

In the remainder of this article, we use $\lambda x \in D.f$ to denote the mathematical function with domain D assigning each $x \in D$ to the corresponding value of expression f . We use $\Phi|_E$ for the restriction of the field evolution Φ to E , defined by $\lambda \epsilon \in E. \Phi(\epsilon)$ for denotational values Φ of local type and by $\lambda \epsilon \in E. \Phi(\epsilon)|_{E^-(\epsilon)}$ for denotational values of field type.

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

$$\begin{aligned}
\mathcal{E}[\![x]\!]_X^E &= X(x)|_E \\
\mathcal{E}[\![\bar{\delta} \mapsto \bar{v}]\!]_X^E &= \lambda \epsilon \in E. \bar{\delta} \mapsto \mathcal{E}[\![\bar{v}]\!]_X^E(\epsilon) \\
\mathcal{E}[\![c(\bar{\ell})]\!]_X^E &= \lambda \epsilon \in E. C[\![c]\!](\mathcal{E}[\![\bar{\ell}]\!]_X^E(\epsilon)) \\
\mathcal{E}[\![b]\!]_X^E &= \lambda \epsilon \in E. \langle b, \mathcal{B}[\![b]\!] \rangle \\
\mathcal{E}[\![d]\!]_X^E &= \lambda \epsilon \in E. \langle d, \lim_n \mathcal{D}[\![d]\!]_n \rangle \\
\mathcal{E}[\![\bar{x}] \Rightarrow^{\tau} e]\!]_X^E &= \lambda \epsilon \in E. \langle \tau, \lambda \bar{\Phi} \in \mathcal{T}[\![\bar{T}]\!]. \mathcal{E}[\![e]\!]_{X \cup \bar{x} \mapsto \bar{\Phi}}^{\text{dom}(\bar{\Phi})} \rangle \\
\mathcal{E}[\![e'(\bar{e})]\!]_X^E &= \lambda \epsilon \in E. \text{snd} \left(\mathcal{E}[\![e']]\!]_X^E(\epsilon) \right) \left(\mathcal{E}[\![\bar{e}]\!]_{X|_{E(e', \epsilon)}}^E(\epsilon) \right) \\
\mathcal{E}[\![\text{nbr}\{e\}]\!]_X^E &= \lambda \epsilon \in E. \lambda \delta \in E^-(\epsilon). \mathcal{E}[\![e]\!]_X^E(\epsilon^\delta) \\
\mathcal{E}[\![\text{rep}(e_1)\{(x) \Rightarrow e_2\}]\!]_X^E &= \lim_n \mathcal{R}[\![\text{rep}(e_1)\{(x) \Rightarrow e_2\}]\!]_n,
\end{aligned}$$

where

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

$$\begin{aligned}
\mathcal{D}[\![d]\!]_0 &= \emptyset \\
\mathcal{D}[\![d]\!]_{n+1} &= \lambda \bar{\Phi} \in \mathcal{T}[\![\bar{T}]\!]. \mathcal{E}[\![\text{body}(d)]\!]_{X \cup \text{args}(d) \mapsto \bar{\Phi}, d \mapsto \lambda \epsilon \in E. \mathcal{D}[\![d]\!]_n}^{\text{dom}(\bar{\Phi})}
\end{aligned}$$

- $E(e', \epsilon)$ is equal to $\{\epsilon' \in E : \text{fst}(\mathcal{E}[\![e']]\!]_X^E(\epsilon')) = \text{fst}(\mathcal{E}[\![e']]\!]_X^E(\epsilon))\}$, that is, to the set of events in E aligned with ϵ with respect to the computation of e' .
- $\mathcal{R}[\![e]\!]_n$ with $e = \text{rep}(e_1)\{(x) \Rightarrow e_2\}$ denotes the rep construct as bounded to n loop steps, and it is defined by rules:

$$\begin{aligned}
\mathcal{R}[\![e]\!]_0 &= \mathcal{E}[\![e_1]\!]_X^E \\
\mathcal{R}[\![e]\!]_{n+1} &= \mathcal{E}[\![e_2]\!]_{X \cup x \mapsto \text{shift}(\mathcal{R}[\![e]\!]_n, \mathcal{R}[\![e]\!]_0)}^E,
\end{aligned}$$

where

$$\text{shift}(\Phi, \Phi_0) = \lambda \epsilon \in \text{dom}(\Phi). \begin{cases} \Phi(\epsilon^-) & \epsilon^- \text{ exists} \\ \Phi_0(\epsilon) & \text{otherwise} \end{cases}$$

“pushes” each value in Φ to the next future event, while falling back to Φ_0 for starting events.

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

¹²We recall that events ϵ are assumed to embed all relevant sensor information.

The denotation of variables is straightforward, while the denotation of constructors and built-in functions is abstracted away assuming that corresponding $C \llbracket c \rrbracket$ and $\mathcal{B} \llbracket b \rrbracket$ are given. To produce neighbouring field values with the correct domain, we require that in case the return type T of b is a *field* type, then $\text{dom}(\mathcal{B} \llbracket b \rrbracket(\bar{\Phi})(\epsilon)) = E^-(\epsilon)$ for all possible $\bar{\Phi}$ in $\mathcal{T} \llbracket T \rrbracket$ of domain E and $\epsilon \in E$. Even though most built-in functions (pure operators, local sensors) could be defined pointwise in the same way constructors are defined, this is not possible for relational sensors (as *nbr-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 are not allowed to occur in source programs, this rule is not necessary to denote such programs.

The denotation of defined functions, as usual, is defined as a *fixpoint* of an iterated process starting from the function $\mathcal{D} \llbracket d \rrbracket_0$ with empty domain. At each subsequent step $n + 1$, the body of d is evaluated with respect to the context that associates the name d itself with the previously obtained function $\mathcal{D} \llbracket d \rrbracket_n$ (and the arguments of d with the respective values). We assume that the resulting function $\mathcal{D} \llbracket d \rrbracket_{n+1}$ is undefined if it calls $\mathcal{D} \llbracket d \rrbracket_n$ with arguments outside of its domain. It follows by easy induction that each such step is a conservative extension, i.e., $\mathcal{D} \llbracket d \rrbracket_n \subseteq \mathcal{D} \llbracket d \rrbracket_{n+1}$, 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 4).

The denotation of a function application $e'(\bar{e})$ is given pointwise by event, and applies the second coordinate of $\mathcal{E} \llbracket e' \rrbracket_X^E(\epsilon)$ (that is, the mathematical function corresponding to e') interpreted in the restricted domain $E(e', \epsilon)$ (computed through the first coordinate of $\mathcal{E} \llbracket e' \rrbracket_X^E$) containing ϵ to the arguments $\mathcal{E} \llbracket \bar{e} \rrbracket_X^E$. Such domain restriction (to devices computing the same function e') is needed to prevent interference among non-aligned devices. 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 $\Phi(\epsilon)$. However, the whole field evolution is actually used since each event in its domain $E(e', \epsilon)$ computes the same function on the same arguments, hence producing the same output field evolution. Thus the rule could also be reformulated as follows:

$$\mathcal{E} \llbracket e'(\bar{e}) \rrbracket_X^E = \bigcup_{\epsilon \in E} \text{snd} \left(\mathcal{E} \llbracket e' \rrbracket_X^E(\epsilon) \right) \left(\mathcal{E} \llbracket \bar{e} \rrbracket_X^E|_{E(e', \epsilon)} \right).$$

The denotation of construct *nbr* yields in each event ϵ a neighbouring field of domain $E^-(\epsilon)$ mapping to the values of expression e in the corresponding events.

The denotation of construct *rep* is carried out by a fixpoint process as for recursive functions. First, a field evolution $\mathcal{R} \llbracket \cdot \rrbracket_0 = \Phi_0$ is computed holding the initial values computed by e_1 in each event. At each subsequent step, the results computed by $\mathcal{R} \llbracket \cdot \rrbracket_n = \Phi$ in each event are made available to their subsequent events through the new assumption $x \mapsto \text{shift}(\Phi, \Phi_0)$ in X . It follows that once the value at each event in $E^-(\epsilon)$ stabilizes, the value at ϵ also stabilizes in one more iteration. Since the events form a well-founded DAG and values at source events (events without predecessor) are steadily equal to the initial value by construction, the whole process stabilizes in each event after a number of iterations at most equal to the (finite) cardinality of its backwards star (see Section 3.1), hence the limit of the process is well defined.

3.4 Example

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

Distance-To. For a first example, consider the expression 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. In formulas, the built-in denotations are as follows:

$$\begin{aligned}
\mathcal{B}[\text{nbr-range}] &= \lambda_.\lambda\epsilon \in \mathbf{E}.\lambda\delta \in \mathbf{E}^-(\epsilon). \text{dist}(\epsilon^\delta, \epsilon) \\
\mathcal{B}[\text{min-hood+}] &= \lambda\Phi \in \mathcal{T}[\text{field}(\text{num})].\lambda\epsilon \in \mathbf{dom}(\Phi). \min(\text{ran}(\Phi(\epsilon) \setminus \delta_\epsilon)) \\
\mathcal{B}[+[f, f]] &= \lambda\bar{\Phi} \in \mathcal{T}[\text{field}(\text{num})].\lambda\epsilon \in \mathbf{dom}(\bar{\Phi}).\lambda\delta \in \mathbf{dom}(\bar{\Phi}(\epsilon)). \\
&\quad \Phi_1(\epsilon)(\delta) + \Phi_2(\epsilon)(\delta) \\
\mathcal{B}[\text{mux}[1, 1, f]] &= \lambda\Phi_1, \Phi_2 \in \mathcal{T}[\text{num}].\lambda\Phi_3 \in \mathcal{T}[\text{field}(\text{num})]. \\
&\quad \lambda\epsilon \in \mathbf{dom}(\bar{\Phi}).\lambda\delta \in \mathbf{dom}(\Phi_3(\epsilon)). \\
&\quad \begin{cases} \Phi_2(\epsilon) & \text{if } \Phi_1(\epsilon) \\ \Phi_3(\epsilon)(\delta) & \text{otherwise,} \end{cases}
\end{aligned}$$

where *dist* is a measure of spatial distance between events.¹³ Notice that the denotation of $+[f, f]$ assumes that the two input neighbouring fields always share a common domain to be able to combine them: We will prove that this property (called *domain alignment*) holds for a broad class of programs (including e_{dist}) in Section 6.1.

Figure 5 shows the evaluation of the denotational semantics for this expression, as evaluated with respect to the neighbourhood graph shown in Figure 4. 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 Φ_s , with corresponding environment $X = s \mapsto \Phi_s$, shown in the top left of Figure 5(b). We assume the devices to be moving¹⁴ so that their relative distance changes over time as depicted in the center left of Figure 5(a).

The outermost component of expression e_{dist} is a `rep` construct, thus $\mathcal{E}[\llbracket e_{\text{dist}} \rrbracket_X]$ is calculated via the following procedure:

- First, $\mathcal{R}[\llbracket e_{\text{dist}} \rrbracket_0]$ is calculated as $\mathcal{E}[\llbracket \infty \rrbracket_X] = \Phi_0$ (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, $\text{shift}(\Phi_0, \Phi_0) = \Phi_0$) and incorporated in the substitution $X_0 = X \cup d \mapsto \Phi_0$. Thus $\mathcal{R}[\llbracket e_{\text{dist}} \rrbracket_1]$ is calculated as $\mathcal{E}[\llbracket e' \rrbracket_{X_0}]$ giving a new field evolution Φ_1 . This evaluation is illustrated step-by-step in Figure 5(a) and (b) top and center, breaking e' into all its subexpressions.
- The process of shift and evaluation is then repeated: Φ_1 is shifted in time (Figure 5(b) bottom left), incorporated in a new substitution $X_1 = X \cup d \mapsto \text{shift}(\Phi_1, \Phi_0)$ and $\mathcal{E}[\llbracket e' \rrbracket_{X_1}]$ is expanded into another field evolution Φ_2 (Figure 5(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 5(c) right).

Notice that due to the characteristics of the `rep` construct, 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` construct 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

¹³We remark that the given denotation of `nbr-range` reflects a possible implementation, where different implementations are also possible.

¹⁴Note that the x -axis in Figure 5 is indexed by *device* and not by *position*.

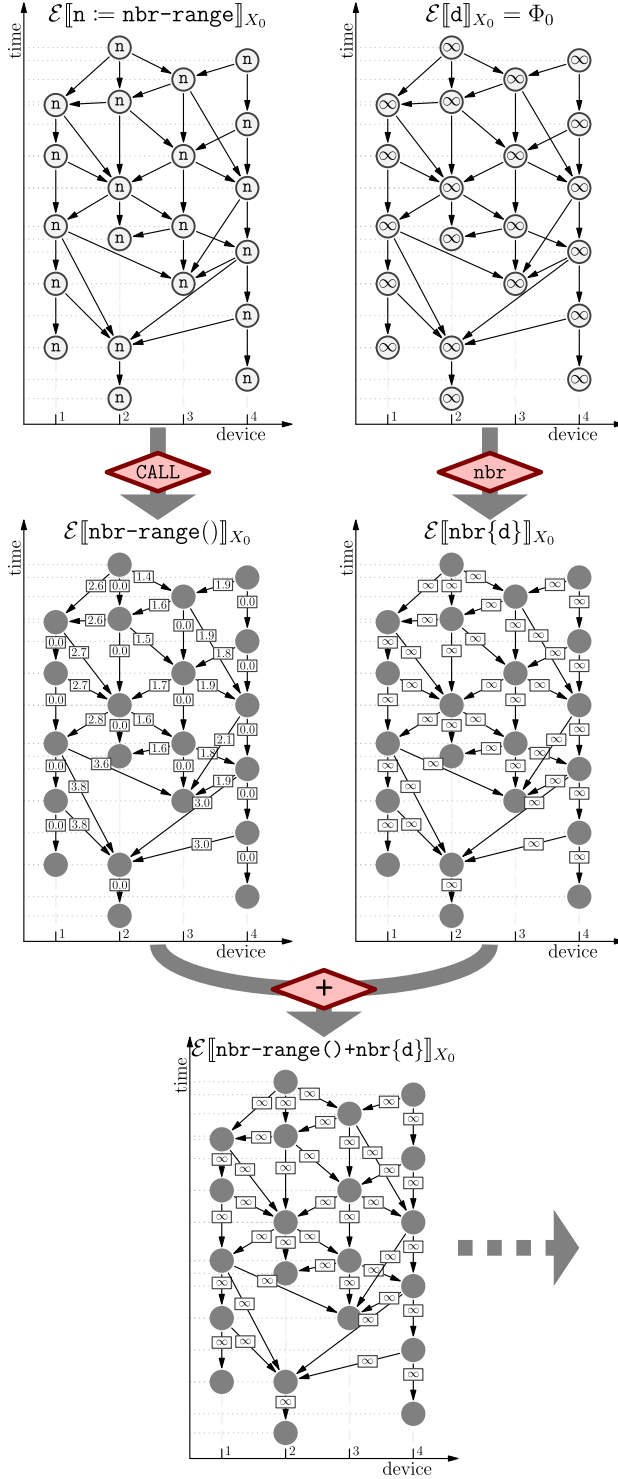


Fig. 5. (a) The denotational semantics of a distance-to calculation.

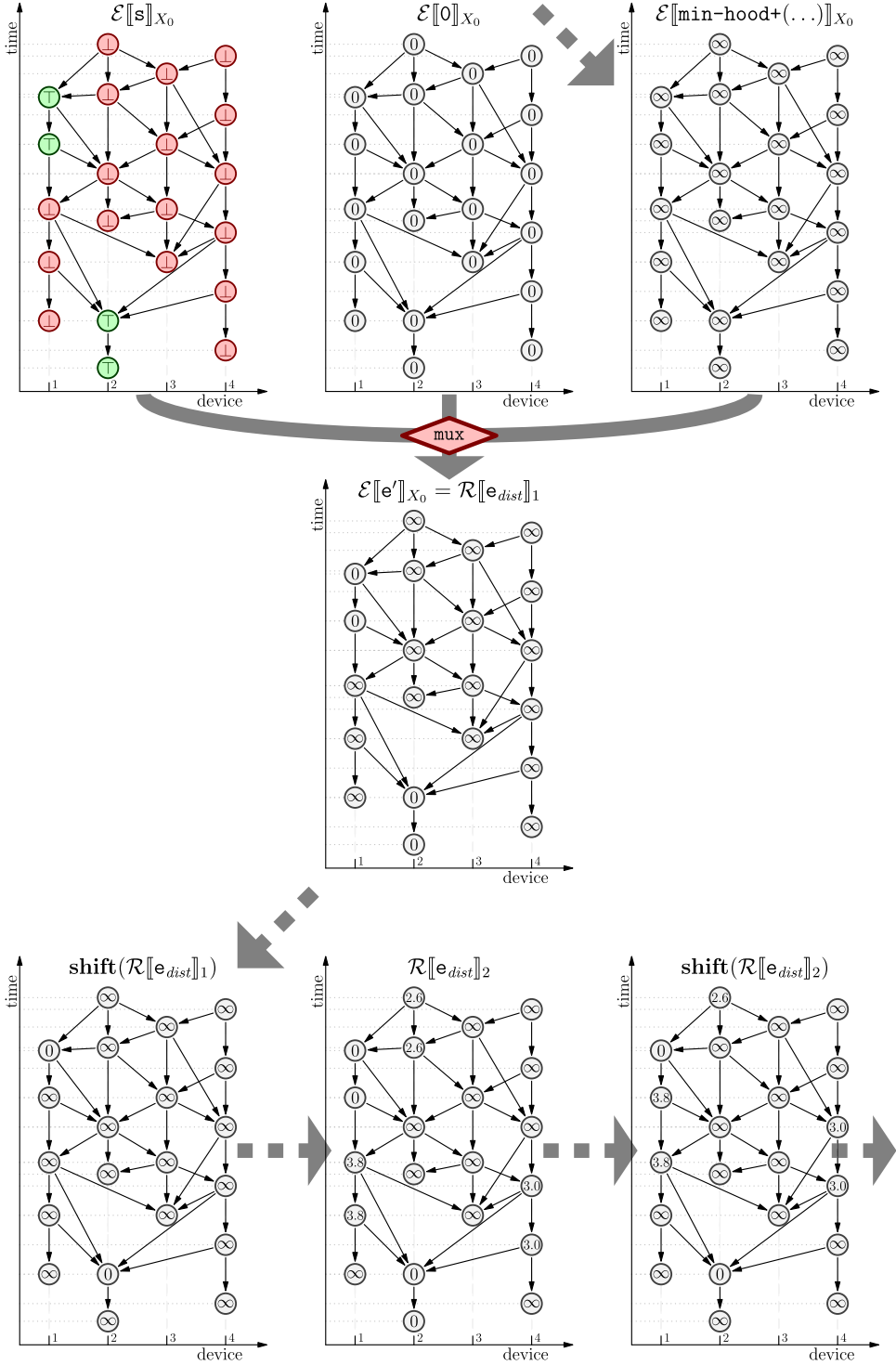


Fig. 5. (b) The denotational semantics of a distance-to calculation (cont.)

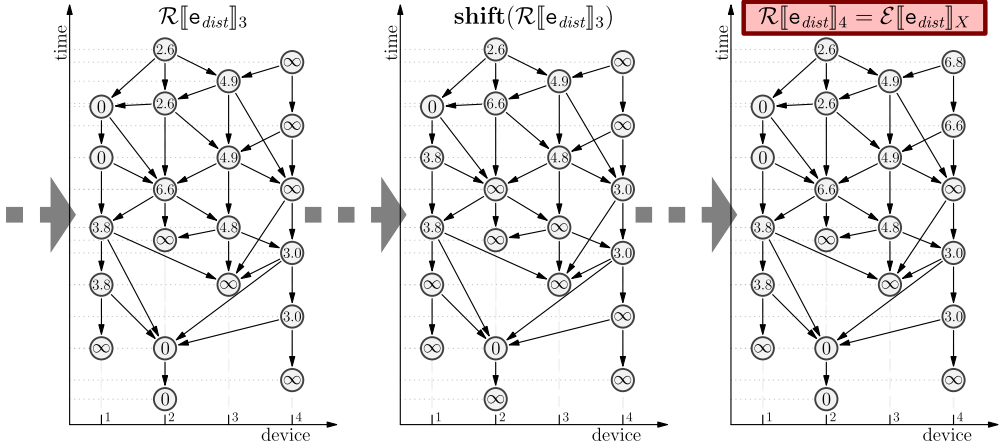


Fig. 5. (c) The denotational semantics of a distance-to calculation (cont.)

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 $e(\bar{e})$ where the functional expression e is *not* constant.

To illustrate branching, we now expand on the previous example by considering the expression e_{avoid} that computes the distance of each device from a given source set *avoiding some obstacles*, that is, computes shortest paths in the network from the source set not containing any node marked as “avoid.”

```
def f(s) { infinity } ;; has type: (bool) → num
def g(s) { e_dist } ;; has type: (bool) → num
mux(avoid, f, g)(source) ;; e_avoid
```

The denotational semantics of e_{avoid} on the sample network in Figure 4 is shown in Figure 6. We consider as input the same source set source (with corresponding field evolution Φ_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 6 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);
- 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.

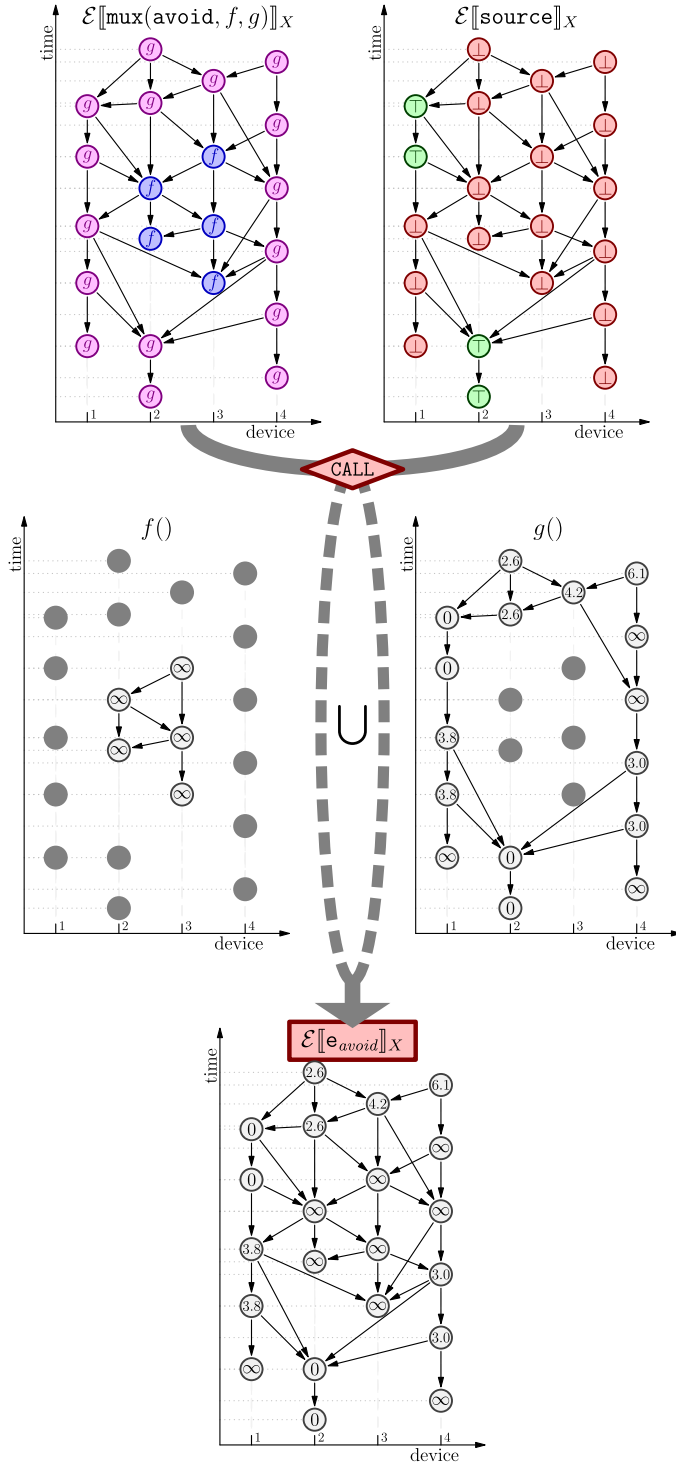


Fig. 6. The denotational semantics of a gradient calculation avoiding obstacles.

Finally, the two different branches are merged together to form the final outcome of the function call, in Figure 6 bottom.

4 OPERATIONAL SEMANTICS

In this section, we introduce an operational semantics for the field calculus. In Section 4.1, we formalize the computation carried out by an individual device at a particular event and then, in Section 4.2, we formalize the computation carried out by a whole network of devices over time.

This semantics can serve as a specification for implementation of programming languages based on the calculus.

4.1 Device Semantics (Big-step Operational Semantics)

According to the “local” viewpoint, individual devices undergo computation in rounds. The *device semantics* models computation of a device within one round, while the *network semantics* (in the next section) describes how the sequence of rounds are related together.

We base operational semantics on the syntax introduced in Section 2.2 (Figure 1). 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 rounds¹⁵ (note that any implementation might massively compress the value-tree, storing only informations about sub-expressions that 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 $\bar{\delta} \mapsto \bar{\theta}$ as short for $\delta_1 \mapsto \theta_1, \dots, \delta_n \mapsto \theta_n$.

The syntax of value-trees and value-tree environments is given in Figure 7 (first frame). Figure 7 (second frame) defines the auxiliary functions ρ and π for extracting the root value and a subtree of a value-tree, respectively (further explanations about function π will be given later); the extension of functions ρ and π to value-tree environments; and the auxiliary functions *name*, *args*, and *body* for extracting the name, formal parameters, and 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 7 (fourth frame). The derived judgements are of the form $\delta; \Theta; \sigma \vdash e \Downarrow \theta$, to be read “expression e evaluates to value-tree θ on device δ with respect to the value-tree environment Θ and sensor state σ ,” 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) σ is a data structure containing enough sensor information to allow each non-pure built-in to be computed; (iv) e is a runtime expression (i.e., an expression that may contain neighbouring field values); (v) 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 e' of e with respect to the value-tree environment Θ' obtained from Θ by extracting the corresponding subtree (when present) in the

¹⁵As formalised by the network semantics (see Section 4.2), when a device δ is shutdown, data from neighbours are lost as well as data from δ itself.

Value-trees and value-tree environments:	
$\theta ::= \vee\langle\bar{\theta}\rangle$	value-tree
$\Theta ::= \bar{\delta} \mapsto \bar{\theta}$	value-tree environment
Auxiliary functions:	
$\rho(\vee\langle\bar{\theta}\rangle) = \vee$ $\pi_i(\vee\langle\theta_1, \dots, \theta_n\rangle) = \theta_i$ if $1 \leq i \leq n$ $\pi^f(\vee\langle\theta_1, \dots, \theta_{n+1}\rangle) = \theta_{n+1}$ if f is a built-in function and $\rho(\theta_{n+1}) = f$ $\pi^f(\vee\langle\theta_1, \dots, \theta_{n+2}\rangle) = \theta_{n+2}$ if f is a non-built-in function and $name(\rho(\theta_{n+1})) = name(f)$ $\pi^f(\theta) = \bullet$ otherwise	
For $aux \in \rho, \pi_i, \pi^f$: $\begin{cases} aux(\bullet) &= \bullet \\ aux(\delta \mapsto \theta, \Theta) &= aux(\Theta) & \text{if } aux(\theta) = \bullet \\ aux(\delta \mapsto \theta, \Theta) &= \delta \mapsto aux(\theta), aux(\Theta) & \text{if } aux(\theta) \neq \bullet \end{cases}$	
$name(d) = d$ $args(d) = \bar{x}$ if $\text{def } d(\bar{x}) \{e\}$ $body(d) = e$ if $\text{def } d(\bar{x}) \{e\}$ $name(\bar{x}) \xrightarrow{\tau} e = \tau$ $args(\bar{x}) \xrightarrow{\tau} e = \bar{x}$ $body(\bar{x}) \xrightarrow{\tau} e = e$	
Syntactic shorthands:	
$\delta; \bar{\pi}(\Theta); \sigma \vdash \bar{e} \Downarrow \bar{\theta}$ where $ \bar{e} = n$ for $\delta; \pi_1(\Theta); \sigma \vdash e_1 \Downarrow \theta_1 \dots \delta; \pi_n(\Theta); \sigma \vdash e_n \Downarrow \theta_n$ $\rho(\bar{\theta})$ where $ \bar{\theta} = n$ for $\rho(\theta_1), \dots, \rho(\theta_n)$ $\bar{x} := \rho(\bar{\theta})$ where $ \bar{x} = n$ for $x_1 := \rho(\theta_1) \dots x_n := \rho(\theta_n)$	
Rules for expression evaluation:	
$\delta; \Theta; \sigma \vdash e \Downarrow \theta$	
$\frac{[E-LOC]}{\delta; \Theta; \sigma \vdash \ell \Downarrow \ell\langle\rangle} \quad \frac{[E-FLD] \quad \phi' = \phi _{\text{dom}(\Theta) \cup \{\delta\}}}{\delta; \Theta; \sigma \vdash \phi \Downarrow \phi'\langle\rangle}$	
$\frac{[E-B-APP] \quad \delta; \bar{\pi}(\Theta); \sigma \vdash \bar{e}, e \Downarrow \bar{\theta}, \theta \quad b = \rho(\theta) \quad v = \langle b \rangle_{\delta}^{\pi^b(\Theta), \sigma}(\rho(\bar{\theta}))}{\delta; \Theta; \sigma \vdash e(\bar{e}) \Downarrow \vee\langle\bar{\theta}, \theta\rangle}$	
$\frac{[E-D-APP] \quad \delta; \bar{\pi}(\Theta); \sigma \vdash \bar{e}, e \Downarrow \bar{\theta}, \theta \quad f = \rho(\theta) \text{ is not a built-in}}{\delta; \pi^f(\Theta); \sigma \vdash body(f)[args(f) := \rho(\bar{\theta})] \Downarrow \theta'}$ $\delta; \Theta; \sigma \vdash e(\bar{e}) \Downarrow \rho(\theta')\langle\bar{\theta}, \theta, \theta'\rangle$	
$\frac{[E-NBR] \quad \Theta_1 = \pi_1(\Theta) \quad \delta; \Theta_1; \sigma \vdash e \Downarrow \theta_1 \quad \phi = \rho(\Theta_1)[\delta \mapsto \rho(\theta_1)]}{\delta; \Theta; \sigma \vdash nbr\{e\} \Downarrow \phi\langle\theta_1\rangle}$	
$\frac{[E-REP] \quad \begin{array}{l} \delta; \pi_1(\Theta); \sigma \vdash e_1 \Downarrow \theta_1 \quad \ell_1 = \rho(\theta_1) \\ \delta; \pi_2(\Theta); \sigma \vdash e_2[x := \ell_0] \Downarrow \theta_2 \quad \ell_2 = \rho(\theta_2) \end{array} \quad \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \text{dom}(\Theta) \\ \ell_1 & \text{otherwise} \end{cases}}{\delta; \Theta; \sigma \vdash rep(e_1)\{x \Rightarrow e_2\} \Downarrow \ell_2\langle\theta_1, \theta_2\rangle}$	

Fig. 7. Big-step operational semantics for expression evaluation.

value-trees in the range of Θ . This process, called *alignment*, is modeled by the auxiliary function π , defined in Figure 7 (second frame). The function π has two different behaviours (specified by its subscript or superscript): $\pi_i(\theta)$ extracts the i th subtree of θ ; and $\pi^f(\theta)$ that

- if f is a built-in function value b , extracts the last subtree of θ , if it is present and its root is b ; and

- if f is a non-built-in function value, extracts the last subtree of θ , if it is present and the root of the second last subtree of θ is a function value with the same name as f .

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 to ensure that domain restriction is obeyed, rule [E-FLD] restricts the domain of the neighbouring field value ϕ to the domain of Θ augmented by δ .

Rule [E-B-APP] models the application of built-in functions. It is used to evaluate expressions of the form $e_{n+1}(e_1 \cdots e_n)$ such that the evaluation of 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 $v\langle\theta_1, \dots, \theta_n, \theta_{n+1}\rangle$, where $\theta_1, \dots, \theta_{n+1}$ are the value-trees produced by the evaluation of the actual parameters and functional expression e_1, \dots, e_{n+1} ($n \geq 0$) and v is the value returned by the function with respect to the current device δ , its sensor state σ and the value-tree environment $\pi^b(\Theta)$ containing only the value-trees associated to the evaluation of the built-in b . Rule [E-B-APP] exploits the special auxiliary function $(\llbracket b \rrbracket)_{\delta}^{\Theta, \sigma}$, whose actual definition is abstracted away, to allow for customised sets of built-in functions. This auxiliary function is such that $(\llbracket b \rrbracket)_{\delta}^{\Theta, \sigma}(\bar{v})$ computes the result of applying built-in function b to values \bar{v} with respect to device δ , sensor state σ and value-tree environment Θ . We require that $(\llbracket b \rrbracket)_{\delta}^{\Theta, \sigma}(\bar{v})$ always yields values of the expected type T where b has a suitable type $(\bar{T}) \rightarrow T \in \mathcal{S}_0(b)$.

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

For example, evaluating the expression $+(1\ 2)$ 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 v as short for the value-tree $v\langle\rangle$. 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 $e_{n+1}(e_1 \cdots e_n)$ such that the evaluation of e_{n+1} produces a value-tree θ_{n+1} whose root $f = \rho(\theta_{n+1})$ is a user-defined function name or an anonymous function value. It is similar to rule [E-B-APP]; however it produces a value-tree that has one more subtree, θ_{n+2} , which is produced by evaluating the body of the function f with respect to the value-tree environment $\pi^f(\Theta)$ containing only the value-trees associated to the evaluation of functions with the same name as f .

To illustrate rule [E-REP] (rep construct), as well as computational rounds, we consider program $\text{rep}(\emptyset)\{(x) \Rightarrow +(x, 1)\}$ (cf. Section 2.2). 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 $\text{rep}(\emptyset)\{(x) \Rightarrow +(x, 1)\}$ we need to evaluate the subexpression $+(0, 1)$, obtained from $+(x, 1)$ by replacing x with 0 , obtaining the value tree $1\langle 0, 1, + \rangle$. This value-tree is then combined with the one obtained from the evaluation of the first argument (which is 0) producing the final value-tree $\theta = 1\langle 0, 1\langle 0, 1, + \rangle \rangle$. 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 $\text{rep}(\theta)\{(x) \Rightarrow +(x, 1)\}$ at the second firing requires 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 $[\text{E-REP}]$, $[\text{E-NBR}]$ we do not assume that Θ is empty whenever it does not contain δ . This might seem unnatural at first glance, since every time a device is rebooted 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 e_{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 $[\text{E-NBR}]$. In this rule, the neighbours' values for e are extracted into a neighbouring field value as $\phi = \rho(\Theta_1)$. Then $\phi(\delta)$ is updated to the more recent value $\ell = \rho(\theta_1)$, as represented by the notation $\phi[\delta \mapsto \ell]$. Consider the program $e' = \text{min-hood}(\text{nbr}\{\text{sns-num}()\})$, 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 $\text{sns-num}()$ on δ_A yields $1\langle\text{sns-num}\rangle$ (by rules $[\text{E-LOC}]$ and $[\text{E-B-APP}]$, since $\langle\text{sns-num}\rangle_{\delta_A}^{0,\sigma}() = 1$); the evaluation of $\text{nbr}\{\text{sns-num}()\}$ on δ_A yields $(\delta_A \mapsto 1)\langle 1\langle\text{sns-num}\rangle \rangle$ (by rule $[\text{E-NBR}]$); and the evaluation of e' on δ_A yields

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

(by rule $[\text{E-B-APP}]$, since $\langle\text{min-hood}\rangle_{\delta_A}^{0,\sigma}(\delta_A \mapsto 1) = 1$). 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

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

respectively. Suppose that device δ_B is the first device that fires a second time. Then the evaluation of 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 $\text{nbr}\{\text{sns-num}()\}$ and $\text{sns-num}()$ is performed, respectively, with respect to the following value-tree environments obtained from Θ_B by alignment:

$$\begin{aligned}\Theta'_B &= \pi_1(\Theta_B) = (\delta_A \mapsto (\delta_A \mapsto 1)\langle 1\langle\text{sns-num}\rangle \rangle, \delta_B \mapsto \dots, \delta_C \mapsto \dots) \\ \Theta''_B &= \pi_1(\Theta'_B) = (\delta_A \mapsto 1\langle\text{sns-num}\rangle, \delta_B \mapsto 2\langle\text{sns-num}\rangle, \delta_C \mapsto 3\langle\text{sns-num}\rangle).\end{aligned}$$

We have that $\langle\text{sns-num}\rangle_{\delta_B}^{\Theta''_B,\sigma}() = 2$; the evaluation of $\text{nbr}\{\text{sns-num}()\}$ on δ_B with respect to Θ'_B yields $\phi\langle 2\langle\text{sns-num}\rangle \rangle$, where $\phi = (\delta_A \mapsto 1, \delta_B \mapsto 2, \delta_C \mapsto 3)$; and $\langle\text{min-hood}\rangle_{\delta_B}^{\Theta_B,\sigma}(\phi) = 1$. Therefore the evaluation of e' on δ_B produces the value-tree $1\langle\phi\langle 2\langle\text{sns-num}\rangle \rangle, \text{min-hood}\rangle$. 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 $\text{pick-hood}(\text{nbr}\{\text{sns-fun}()\})$, 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 $\ell_0 = () \xRightarrow{\tau_0} 0$ on

System configurations and action labels:		
Ψ	$::= \bar{\delta} \mapsto \bar{\Theta}$	status field
τ	$::= \bar{\delta} \mapsto \bar{I}$	topology
Σ	$::= \bar{\delta} \mapsto \bar{\sigma}$	sensors-map
Env	$::= \tau, \Sigma$	environment
N	$::= \langle Env, \Psi \rangle$	network configuration
act	$::= \delta \mid env$	action label
Environment well-formedness:		
$WFE(\tau, \Sigma)$ holds iff $\text{dom}(\tau) = \text{dom}(\Sigma)$ and $\tau(\delta) \subseteq \text{dom}(\Sigma)$ for all $\delta \in \text{dom}(\Sigma)$.		
Transition rules for network evolution:		$N \xrightarrow{act} N$
$\frac{[N\text{-FIR}] \quad Env = \tau, \Sigma \quad \tau(\delta) = \bar{\delta} \quad \delta; F(\Psi)(\delta); \Sigma(\delta) \vdash e_{\text{main}} \Downarrow \theta \quad \Psi_1 = \bar{\delta} \mapsto \{\delta \mapsto \theta\}}{\langle Env, \Psi \rangle \xrightarrow{\delta} \langle Env, F(\Psi)[\Psi_1] \rangle}$		
$\frac{[N\text{-ENV}] \quad WFE(Env') \quad Env' = \tau, \bar{\delta} \mapsto \bar{\sigma} \quad \Psi_0 = \bar{\delta} \mapsto \emptyset}{\langle Env, \Psi \rangle \xrightarrow{env} \langle Env', \Psi_0[\Psi] \rangle}$		

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

δ_A and δ_B , and returns $\ell_1 = () \xRightarrow{\tau_1} e'$ on δ_C , where $e' = \text{min-hood}(\text{nbr}\{\text{sns-num}()\})$ 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

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

where the root of the first subtree of θ'_A is the anonymous function value 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

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

respectively, where θ_C is the value-tree for 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

$$1\langle \ell_1\langle \phi'\langle \ell_1\langle \text{sns-fun} \rangle, \text{pick-hood} \rangle, \theta'_A \rangle,$$

where $\phi' = (\delta_A \mapsto \ell_1, \delta_C \mapsto \ell_1)$ and $\theta''_A = 1\langle (\delta_A \mapsto 1, \delta_C \mapsto 3)\langle \text{sns-num} \rangle, \text{min-hood} \rangle$, since, according to rule [E-D-APP], the evaluation of the body 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 value 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 .

4.2 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 8 (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) sensor state (as described in the previous section). 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 8 (bottom), rule $[\text{N-FIR}]$, is a given filtering operation meant to clear out old stored values from the value-tree environments in Ψ , usually based on space/time tags attached to value-trees: an example of such a filter is given in Example 4.1. We use the following notation for status fields. Let $\bar{\delta} \mapsto \Theta$ denote the map sending each device identifier in $\bar{\delta}$ to the same value-tree environment Θ . Let $\Theta_0[\Theta_1]$ denote the value-tree environment with domain $\text{dom}(\Theta_0) \cup \text{dom}(\Theta_1)$ 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 $N \xrightarrow{\text{act}} N'$, 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 8 (bottom). Rule $[\text{N-FIR}]$ 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. Recall that the local sensors $\Sigma(\delta)$ are used by the auxiliary function $(\llbracket \cdot \rrbracket)_{\delta}^{\Theta, \sigma}$ that gives the semantics to the built-in functions. Rule $[\text{N-ENV}]$ takes into account the change of the environment to a new well-formed environment Env' . Let $\bar{\delta}$ 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.

Remark 8 (On device shutdown and boot). The shutdown and the boot of a device δ are modelled by two applications of rule $[\text{N-ENV}]$: one deleting δ from the domain of the status field and another inserting it. Note that when a device δ is shutdown, data from neighbours is lost as well as data from δ itself.

Example 4.1. 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 $\bar{\delta}$ to $\bar{F}(\Theta)$, where

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

(recall that t_d is the decay parameter).

Example 4.2 (A continuation of Example 4.1 that requires some notions given in Section 3.1). Furthermore, we can proceed in analogy with Example 3.1 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 the following:

- an application of Rule $[\text{N-ENV}]$ updating with the topology τ given by

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

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

- an application 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 application of Rule $[N-FIR]$ on device δ_ϵ .

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

5 REFINED TYPING

In this section, we present a variant of the Hindley-Milner type system [21] for the proposed calculus, that is, a polymorphic refinement of the one presented in Section 2, useful to type non-trivial specifications.

This type system has four main kinds of types and is designed specifically 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 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` and function application: namely, whenever two neighbouring field values are combined in an expression e , domain alignment grants that they share the same domain (consisting of the current device and the neighbours that have calculated the same expression e). Moreover, domain alignment is required 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. For a formal definition of domain alignment, see Theorem 6.1.

This section is organized as follows: First, Section 5.1 introduces a refinement of the monomorphic type system given in Section 2.4, whose peculiar features are then illustrated in Section 5.2 by means of examples; then, Section 5.3 presents the variant of the Hindley-Milner type system for HFC and points out that it corresponds to the refined monomorphic type system of Section 5.1.

5.1 A Refined Monomorphic Type System for HFC

We now present a refined monomorphic type system, which extends the one in Figure 2 with two additional categories of types (*return types* and *local return types*) and several additional restrictions in the typing rules.

The syntax of types is given in Figure 9 (top), together with a graphical representation of their inclusion relations. A *type* T is either a local type or a field type. A *local type* L is either a built-in type B (numbers, Booleans, pairs, lists etc.), or the type of a function $(\bar{T}) \rightarrow R$ (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 local return type, or a field type. Notice that these types are the only ones allowed to appear on the right-hand side of the \rightarrow operator, while any type is allowed on the left-hand side. A *local return type* S is either a built-in type B or the type of a function $(\bar{T}) \rightarrow S$ (possibly of arity zero). A *field type* F is the type `field(S)` of a field whose range contains values of local return type S . The type system does not contemplate types of the kind $(\dots) \rightarrow \dots \rightarrow (\dots) \rightarrow F$ (functions that return functions that return neighbouring field values), since expressions involving such types can be unsafe (as exemplified in the next subsection).

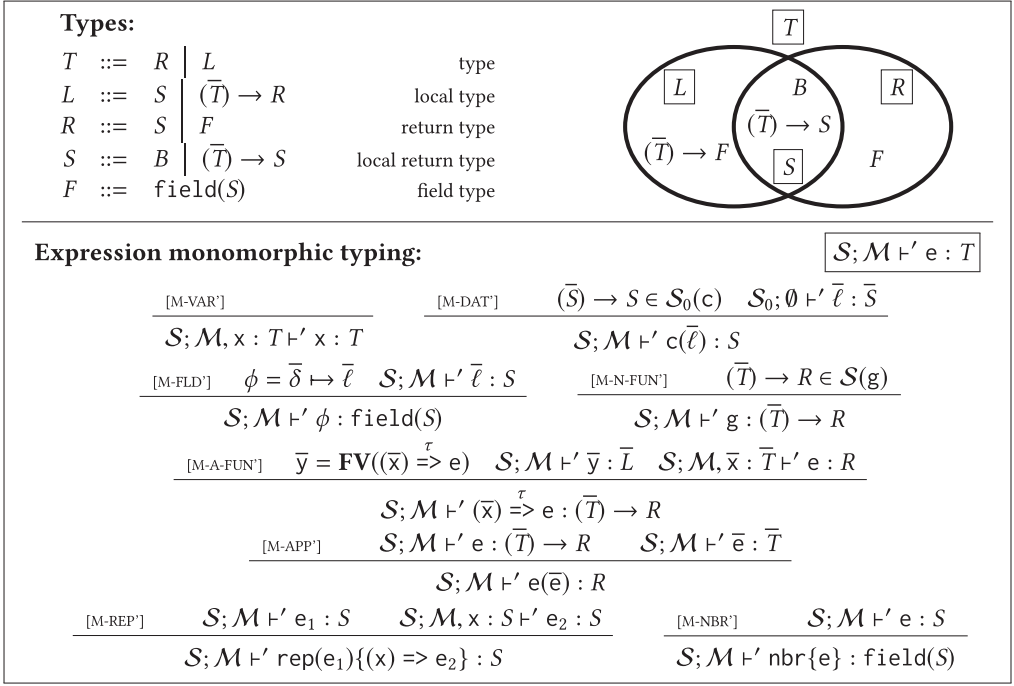


Fig. 9. Refined monomorphic typing for HFC expressions.

The type rules are given in Figure 9 (bottom). The typing judgement for expressions is of the form “ $S; \mathcal{M} \vdash e : T$ ”, to be read: “ e has type T under the type-set assumptions S (for data constructors, built-in and defined functions) and the type assumptions \mathcal{M} ” (for the program variables occurring in e).

Rules [M-VAR’], [[M-DAT’], [M-FLD’], [M-N-FUN’], and [M-APP’] are analogous to the corresponding rules in Figure 2, except that constructors are assumed to operate on local return types $c : (\bar{S}) \rightarrow S$, field values are built from local return types $\bar{\ell} : S$, and functions are assumed to have a legal return type $f : (\bar{T}) \rightarrow R$.

Rule [M-A-FUN’] (for anonymous functions) ensures that anonymous function expressions $(\bar{x}) \xRightarrow{\tau} e$ have return type in R and do not contain free variables of field type to avoid domain alignment errors (as will be exemplified in Section 5.2).

Rule [M-REP’] (for rep-expressions) ensures that both the variable x , its initial value e_1 and the body e_2 have (the same) local return type. In fact, allowing field types might produce domain mismatches, while a rep-expression of type $(\bar{T}) \rightarrow F$ would be a non-constant (thus not safely applicable) function returning neighbouring field values.

Rule [M-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) or a “neighbouring field ϕ of functions returning neighbouring field values” (as there would be no type-safe way to produce or use a non-constant such ϕ).

Notice that the present system can be extended to cover defined functions and whole programs by means of rules analogous to those in Section 2.4.2.

$\frac{[M\text{-FUNCTION}'] \quad S, d : \{\bar{T}^{(1)} \rightarrow R^{(1)}\}; \bar{x} : \bar{T}^{(1)} \vdash' e : R^{(1)} \cdots S, d : \{\bar{T}^{(m)} \rightarrow R^{(m)}\}; \bar{x} : \bar{T}^{(m)} \vdash' e : R^{(m)} \quad m \geq 1}{S \vdash' \text{def } d(\bar{x}) \{e\} : \{\bar{T}^{(1)} \rightarrow R^{(1)}, \dots, \bar{T}^{(m)} \rightarrow R^{(m)}\}}$	
$\frac{[M\text{-PROGRAM}'] \quad F_i = (\text{def } d_i(_) _) \quad S_{i-1} \vdash' F_i : Lset_i \quad S_i = S_{i-1}, d_i : Lset_i \quad (i \in 1 \dots n) \quad S_n; \emptyset \vdash' e : T}{S_0 \vdash' F_1 \cdots F_n e : T}$	

Since each of those rules is a restriction of a corresponding rule in Figure 2, the present type system gives to any expression a subset of the types given by the type system in Section 2.4.1.

THEOREM 5.1 (SYSTEM \vdash' REFINES SYSTEM \vdash). *If $S_0 \vdash' P : T$, then $S_0 \vdash P : T$.*

PROOF. See Appendix A.1. □

5.2 Examples

The type system in the previous section enforces some peculiar restrictions, designed to prevent ill-behaving programs to be typed. We now clarify these peculiar features by means of a 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 Reference [56] 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 3), thus similar issues apply to functions returning neighbouring field values. Consider the expression e_{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 $f = (x) \Rightarrow x$ is obtained whence applied to `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 `nbr{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 e_{safe} :

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

In this case, on every device the same function $f = (x) \Rightarrow x$ is obtained whence applied to `nbr{0}`, that is, no alignment is required thus the function application returns the whole neighbouring field value $\phi = \text{nbr}\{0\}$, which can be safely combined with `nbr{uid}`. This suggests 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 5 ensures this distinction. Besides performing standard checks, the type system performs the following additional checks 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 that 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 e_{wrong} according to Remark 3, then 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 $() \rightarrow \text{field}(\text{num}) \rightarrow \text{field}(\text{num})$.

- *In an anonymous function expression $(\bar{x}) \stackrel{\tau}{\Rightarrow} e$, the free variables \bar{y} of e that are not in \bar{x} have local type.* This prevents a device δ from creating a closure $e' = (\bar{x}) \stackrel{\tau}{\Rightarrow} e[\bar{y} := \bar{\phi}]$ containing neighbouring field values $\bar{\phi}$ (whose domain is by construction equal to the subset of the aligned neighbours of δ). The closure 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 e' to some arguments); and the evaluation of the body of e' may involve use of a neighbouring field value ϕ in $\bar{\phi}$ such that the set of aligned neighbours of δ' is different from the domain of ϕ . For instance, the expression e'_{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 $() \Rightarrow \text{min-hood}(x + \text{nbr}\{0\})$ 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

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

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 e'_{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 e'_{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 $\text{rep}(e_1)\{(x) \Rightarrow e_2\}$ it holds that x , e_1 and 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 e''_{wrong} :

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

that should have type num, is ill typed.

- In an nbr-expression $\text{nbr}\{e\}$ 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.

5.3 Hindley-Milner Typing for HFC

In this section, we present a variant of the Hindley-Milner type system that corresponds to the refined monomorphic type system introduced in Section 5.1.

The syntax of types and type schemes is given in Figure 10 (top). Types are the same as in the refined monomorphic typing, with the addition of four kinds of type variables s , r , l , t —similarly as the Standard ML type system features two kinds (equality and non-equality types [42]). These different kinds allow functions to behave polymorphically while enforcing the same ad-hoc restrictions of the monomorphic type system (necessary to guarantee type preservation and domain alignment), as shown in Example 5.2. *Local type schemes*, ranged over by LS , support typing polymorphic uses of data constructors, built-in functions and user defined-functions. Namely, for each data constructor, built-in function or user-defined function g there is a *local type scheme* $\forall \bar{t} \bar{l} \bar{r} \bar{s}. L$, where \bar{t} , \bar{l} , \bar{r} , and \bar{s} are all the type variables occurring in the type L , respectively. Each use of g can be typed with any type obtained from $\forall \bar{t} \bar{l} \bar{r} \bar{s}. L$ by replacing the type variables \bar{t} with types, \bar{l} with local types, \bar{r} with return types and \bar{s} with local return types.

Type environments, ranged over by \mathcal{A} , collect type assumptions for program variables, while *Local-type-scheme environments*, ranged over by \mathcal{D} and written $\bar{g} : \bar{LS}$, collect the local type schemes for the data constructors and built-in functions together with the local type schemes inferred for the user-defined functions. In particular, the distinguished *built-in local-type-scheme environment* \mathcal{D}_0 associates a local type scheme to each data constructor c and built-in function b —Figure 11 shows the local type schemes for the data constructors and built-in functions used throughout this article (corresponding the set of monomorphic types given in Figure 3).

The typing rules are given in Figure 10 (bottom). The typing judgement for expressions is of the form “ $\mathcal{D}; \mathcal{A} \vdash e : T$ ”, to be read: “ e has type T under the local-type-scheme assumptions \mathcal{D} and the type assumptions \mathcal{A} .” Note that the type rules are syntax directed, so they straightforwardly describe a type inference algorithm, and that there is no need for a rule typing neighbouring field expressions ϕ , since they are not allowed to appear in source programs.

All the expression typing rules except Rule $T\text{-DAT}$ and Rule $T\text{-N-FUN}$ are analogous to the corresponding ones in the refined monomorphic type system (Figure 9). Rule $T\text{-N-FUN}$ (for built-in and user-defined function names) ensures that the local type scheme $\mathcal{D}(g) = \forall \bar{t} \bar{l} \bar{r} \bar{s}. L$ associated to the built-in 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 $T\text{-DAT}$ (for data values) allows us to assign to the data constructor c any instance of the type scheme type $\mathcal{D}_0(c) = \forall \bar{t} \bar{l} \bar{r} \bar{s}. L$ that meets the types of the given arguments, which need to be (local) *values*.

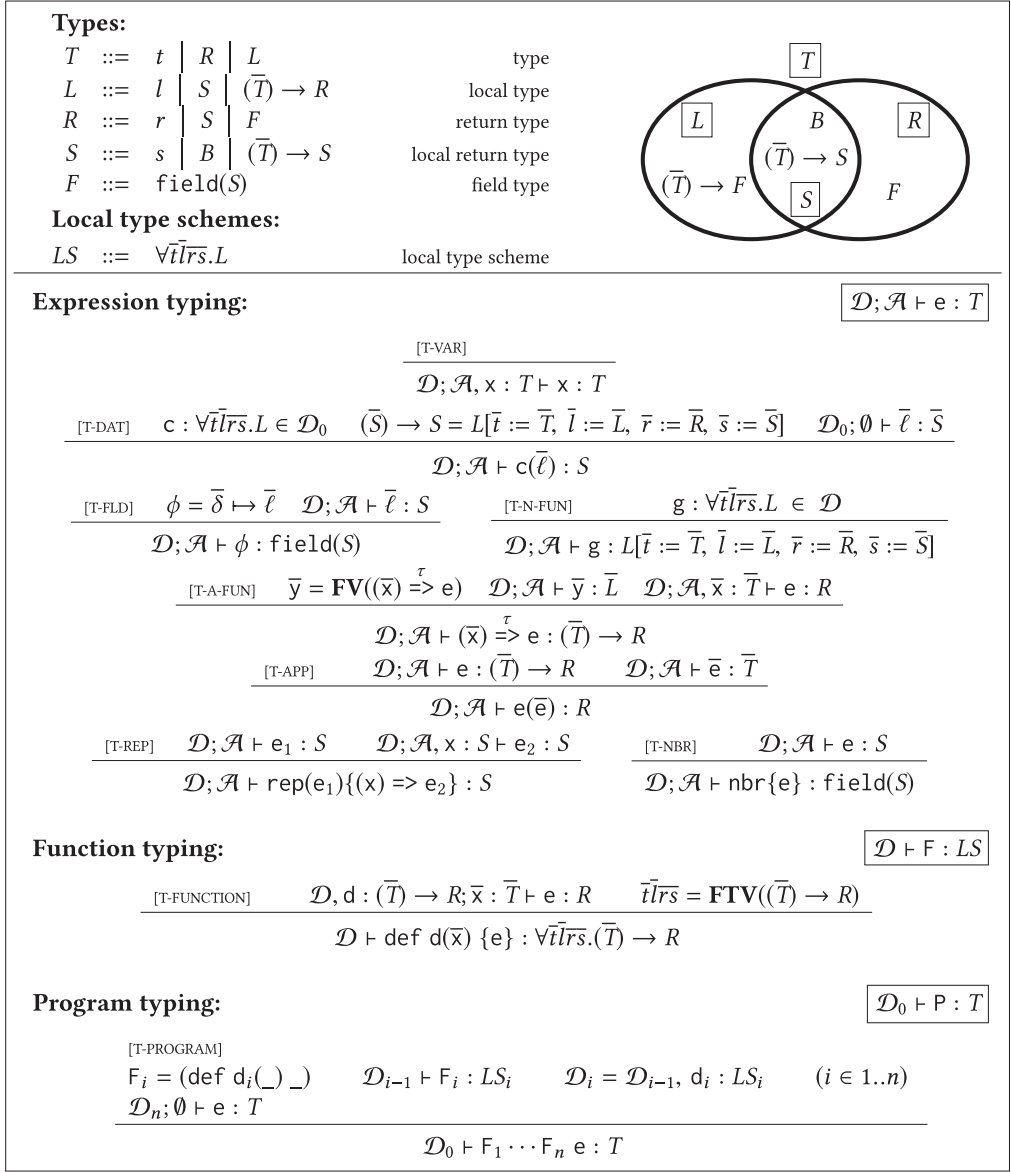


Fig. 10. Hindley-Milner typing for HFC expressions, function declarations, and programs.

Function declaration typing (represented by judgement “ $\mathcal{D} \vdash F : LS$ ”) and program typing (represented by judgement “ $\mathcal{D}_0 \vdash P : T$ ”) are almost standard and, for what concerns the refinement, analogous to those defined in Section 5.1. We say that a program P is *well typed* to mean that $\mathcal{D}_0 \vdash P : T$ holds for some type T .

Remark 9 (On automatic type inference). Since the type system in Figure 10 is a customisation of the Hindley-Milner type system [21] to the field calculus, there is an algorithm (not presented here) that, given an expression e , type assumptions for its free variables, and type scheme assumptions for data constructors, built-in and user-defined functions: either fails (if the expression cannot be

Built-in constructors:

$\mathcal{B}(\text{True})$	$= () \rightarrow \text{bool}$
$\mathcal{B}(\text{False})$	$= () \rightarrow \text{bool}$
$\mathcal{B}(0)$	$= () \rightarrow \text{num}$
$\mathcal{B}(\text{Pair})$	$= \forall s_1 s_2. (s_1, s_2) \rightarrow \text{pair}(s_1, s_2)$
$\mathcal{B}(\text{Nil})$	$= \forall s. () \rightarrow \text{list}(s)$
$\mathcal{B}(\text{Cons})$	$= \forall s. (s, \text{list}(s)) \rightarrow \text{list}(s)$

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

$\mathcal{B}(\text{fst})$	$= \forall s_1 s_2. (\text{pair}(s_1, s_2)) \rightarrow s_1$
$\mathcal{B}(\text{snd})$	$= \forall s_1 s_2. (\text{pair}(s_1, s_2)) \rightarrow s_2$
$\mathcal{B}(\text{pair})$	$= \forall s_1 s_2. (s_1, s_2) \rightarrow \text{pair}(s_1, s_2)$
$\mathcal{B}(\text{head})$	$= \forall s. (\text{list}(s)) \rightarrow s$
$\mathcal{B}(\text{tail})$	$= \forall s. (\text{list}(s)) \rightarrow \text{list}(s)$
$\mathcal{B}(\text{min-hood}, \text{min-hood}+)$	$= \forall s. (\text{field}(s)) \rightarrow s$
$\mathcal{B}(\text{pick-hood})$	$= \forall s. (\text{field}(s)) \rightarrow s$
$\mathcal{B}(\text{map-hood})$	$= \forall \bar{s}. \forall s' ((\bar{s}) \rightarrow s', \text{field}(\bar{s})) \rightarrow \text{field}(s')$
$\mathcal{B}(\text{fold-hood})$	$= \forall s. ((s, s) \rightarrow s, \text{field}(s)) \rightarrow s$
$\mathcal{B}(\text{mux})$	$= \forall s. (\text{bool}, s, s) \rightarrow s$
$\mathcal{B}(\text{and}, \text{or})$	$= (\text{bool}, \text{bool}) \rightarrow \text{bool}$
$\mathcal{B}(+, -, *, /)$	$= (\text{num}, \text{num}) \rightarrow \text{num}$
$\mathcal{B}(=)$	$= \forall t. (t, t) \rightarrow \text{bool}$

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

$\mathcal{B}(\text{sns-range})$	$= () \rightarrow \text{num}$
$\mathcal{B}(\text{sns-injection-point})$	$= () \rightarrow \text{bool}$
$\mathcal{B}(\text{sns-injected-function})$	$= () \rightarrow (() \rightarrow \text{num})$
$\mathcal{B}(\text{nbr-range})$	$= () \rightarrow \text{field}(\text{num})$
$\mathcal{B}(\text{uid})$	$= () \rightarrow \text{num}$

Fig. 11. Local type schemes for the data constructors and the built-in functions used throughout this article.

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 a *unification* routine as in Reference [40], 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).

Example 5.2 (Parametric Types). Consider the following variants of an “apply” function with their corresponding principal types, showing how the range of a type variable is automatically tuned depending on the contexts in which it is used.

$(x, y) \Rightarrow x(y) :$	$\forall tr. ((t) \rightarrow r, t) \rightarrow r$
$(x, y) \Rightarrow x(y + 1) :$	$\forall r. ((\text{num}) \rightarrow r, \text{num}) \rightarrow r$
$(x, y) \Rightarrow ((z) \Rightarrow x(y)) :$	$\forall tlr. ((l) \rightarrow r, l) \rightarrow ((t) \rightarrow r)$
$(x, y) \Rightarrow \text{nbr}\{x(y)\} :$	$\forall ts. ((t) \rightarrow s, t) \rightarrow \text{field}(s)$
$(x, y) \Rightarrow \text{rep}(y)\{(z) \Rightarrow x(z)\} :$	$\forall s. ((s) \rightarrow s, s) \rightarrow s$

Derivations of these typing judgements can be straightforwardly obtained since the type system is syntax directed.

The present type system corresponds closely to the refined monomorphic type system presented in Section 5.1, as they assign the same monomorphic types to expressions and programs. To formally state this result, we first introduce a convent auxiliary notation: Given a local type scheme $LS = \forall \overline{tlrs}. L$, the set of the *monomorphic instances* of LS , denoted by $\mathbf{MS}(LS)$, is the set of

monomorphic types defined as follows:

$$\begin{aligned} \mathbf{MS}(\forall \bar{l} \bar{r} \bar{s}. L) = \{L[\bar{l} := \bar{T}, \bar{l} := \bar{L}, \bar{r} := \bar{R}, \bar{s} := \bar{S}] \mid & \bar{T} \text{ monomorphic types,} \\ & \bar{L} \text{ monomorphic local types,} \\ & \bar{R} \text{ monomorphic return types,} \\ & \bar{S} \text{ monomorphic local return types}\}. \end{aligned}$$

Notice that $\mathbf{MS}(LS)$ is infinite whenever LS is not monomorphic. Thus, we cannot assume that a monomorphic environment \mathcal{S} corresponds to a type-scheme environment \mathcal{D} for user-defined functions. However, we can assume the correspondence holds for built-in functions, as their type-sets are allowed to be infinite.

THEOREM 5.3 (HINDLEY-MILNER TYPING \vdash VS MONOMORPHIC TYPING \vdash'). *Assume that \mathcal{S}_0 and \mathcal{D}_0 have the same domain and for each g in their domain it holds that $\mathcal{S}_0(g) = \mathbf{MS}(\mathcal{D}_0(g))$. Then, for any monomorphic type T , it holds that $\mathcal{D}_0 \vdash P : T$ if and only if $\mathcal{S}_0 \vdash' P : T$.*

PROOF. See Appendix A.2. □

This above result can be combined with Theorem 5.1 to obtain that any monomorphic type assigned to an expression e or program P by the present type system can also be assigned to e or P by the type system in Section 2.4.

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) computational adequacy and 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 remark that throughout this article we assume that the execution of every firing event ϵ terminates (cf. Remark 4) and is instantaneous and that every event happens at a distinct moment in time.

6.1 Device Computation Type Preservation and Domain Alignment

Here we formally state the device computation type preservation and domain alignment properties (cf. Section 5) for the HFC calculus. To state these properties we introduce the notion of well-formed value-tree environment for an expression.

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

- if e is a value, θ is of the form $v\langle \rangle$;
- if $e = \text{nbr}\{e_1\}$, θ is of the form $v\langle \theta_1 \rangle$;
- if $e = \text{rep}(e_1)\{(x) \Rightarrow e_2\}$, θ is of the form $v\langle \theta_1, \theta_2 \rangle$ where θ_2 is well-formed for e_2 with the additional assumption $x : T$;
- if $e = e_{n+1}(\bar{e})$, θ is of one of the following two forms:
 - $v\langle \bar{\theta}, \theta_{n+1} \rangle$ where $f = \rho(\theta_{n+1})$ is a built-in function,
 - $v\langle \bar{\theta}, \theta_{n+1}, \theta_{n+2} \rangle$ where f is not a built-in function and θ_{n+2} is well-formed with respect to $e_{n+2} = \text{body}(f)$ with the additional assumptions that $\text{args}(f) : \bar{T}'$ where $\bar{e} : \bar{T}'$.

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

As these notions are defined, we can now formally state the type preservation and domain alignment properties (cf. Section 5).

THEOREM 6.1 (DEVICE COMPUTATION TYPE PRESERVATION AND DOMAIN ALIGNMENT). *Let $\mathcal{A} = \bar{x} : \bar{T}$, $\mathcal{D}; \emptyset \vdash \bar{v} : \bar{T}$, so that $\text{length}(\bar{v}) = \text{length}(\bar{x})$. If $\mathcal{D}; \mathcal{A} \vdash e : T$, $\Theta \in WFVTE(\mathcal{D}, \mathcal{A}, e)$ and $\delta; \Theta; \sigma \vdash e[\bar{x} := \bar{v}] \Downarrow \theta$, then:*

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

PROOF. See Appendix A.3. □

6.2 Computational Adequacy and Abstraction

We are now able to prove that the denotational semantics introduced in Section 3 is computationally adequate and satisfies a form of abstraction, that we call computational abstraction, with respect to the operational semantics introduced in Section 4. We prove these properties for programs that are well typed with respect to type system given in Section 5—because of Theorem 5.3 we can consider only monomorphic types for expressions.

The notions of adequacy and abstraction are presented in literature in many (slightly) different forms (see, e.g., References [20, 52]), none of them fitting without modifications into the setting of HFC. According to Reference Curien [20], given a typed sequential language admitting function types and function application, we say that:

- The denotational semantics is *computationally adequate* iff for any closed expression e and value v of *observable* type T , e evaluates to v (operationally) if and only if $\llbracket e \rrbracket = \llbracket v \rrbracket$ (denotationally).
- Two possibly non-closed expressions e_1, e_2 are *operationally equivalent* (in formulas, $e_1 =_{op} e_2$) iff for all contexts C (terms with a hole) such that $C[e_i]$ has observable type, $C[e_1]$ evaluates to v if and only if $C[e_2]$ evaluates to v .
- *Full abstraction* holds iff given any two possibly non-closed expressions e_1, e_2 , their denotations coincide if and only if they are operationally equivalent:

$$\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket \quad \Leftrightarrow \quad e_1 =_{op} e_2.$$

The left-to-right direction is called *adequacy* and is a consequence of computational adequacy for compositional semantics, whereas the other direction is the hard one.

These notions need to be adjusted for the field calculus to accommodate for the influence of the environment and of the previous rounds of computations. Idea for achieving this is to add a “for all environments” quantifier inside any operational or denotational statement.

Consider a program e_{main} interpreted both

- operationally through a sequence of transitions that we refer as *operational evolution*;
- denotationally through a set of events (with their associated information and *neighbours* function) and corresponding constructor and built-in interpretation functions $C[\llbracket \cdot \rrbracket], \mathcal{B}[\llbracket \cdot \rrbracket]$, which we refer as *denotational environment*.

Assume that each event ϵ is in bijection with one occurrence of a transition $\xrightarrow{\delta_\epsilon}$. Let Ψ_ϵ , τ_ϵ , Σ_ϵ denote the value of the status field, topology, and sensor map just before the occurrence corresponding to ϵ . Let $\Theta_\epsilon = F(\Psi_\epsilon)(\delta_\epsilon)$, $\sigma_\epsilon = \Sigma_\epsilon(\delta_\epsilon)$ denote the value-tree environment and sensor state 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 E , $\Theta_\epsilon = \{\delta_{\epsilon'} \mapsto \theta_{\epsilon'} : \text{neigh}(\epsilon, \epsilon')\}$. This is equivalent to the assertion that for each ϵ, ϵ' in E , $\text{neigh}(\epsilon, \epsilon')$ holds if and only if:
 - $\delta_\epsilon \in \tau_{\epsilon'}(\delta_{\epsilon'})$;
 - there is no further ϵ'' between ϵ' and ϵ such that $\delta_{\epsilon'} = \delta_{\epsilon''}$ and $\delta_\epsilon \in \tau_{\epsilon''}(\delta_{\epsilon''})$;
 - $F(\Psi_\epsilon)$ does not filter out ϵ' (referring to Example 4.1: $t_{\epsilon'} \geq t_\epsilon - t_d$).
- (2) $C\llbracket c \rrbracket$ is such that $\mathcal{E}\llbracket c(\bar{\ell}) \rrbracket^E(\epsilon) = C\llbracket c \rrbracket(\mathcal{E}\llbracket \bar{\ell} \rrbracket^E(\epsilon))$ for all $E \subseteq E$ and $\epsilon \in E$.
- (3) $\mathcal{B}\llbracket b \rrbracket$ operates pointwise on its arguments (i.e., does not incorporate communication between events) and correctly translates the behaviour of $\llbracket b \rrbracket_{\delta_\epsilon}^{\Theta_\epsilon, \sigma_\epsilon}$, that is:

$$\mathcal{E}\llbracket \llbracket b \rrbracket_{\delta_\epsilon}^{\Theta_\epsilon, \sigma_\epsilon}(\bar{v}) \rrbracket^E(\epsilon) = \mathcal{B}\llbracket b \rrbracket(\mathcal{E}\llbracket \bar{v} \rrbracket^E(\epsilon))$$

for all values \bar{v} of the correct type and $E = \{\epsilon' \in E^-(\epsilon) : \delta_{\epsilon'} \in \text{dom}(\Theta)\}$.¹⁶

We remark that the possible implementations outlined in Examples 3.1 and 4.1 are coherent. Given this coherence condition, we can prove that computational adequacy holds in its strongest possible form, where every type is assumed to be observable.

THEOREM 6.2 (COMPUTATIONAL ADEQUACY). *Assume that the denotational environment is coherent with the operational evolution of the network and e is well typed (that is, $\mathcal{D}; \emptyset \vdash e : T$).*

Then $\mathcal{E}\llbracket e \rrbracket^E(\epsilon) = \mathcal{E}\llbracket v_e \rrbracket^E(\epsilon)$ for each ϵ in E , where $v_e = \rho(\theta_\epsilon)$ is the operational outcome of expression e in fire ϵ .

PROOF. See Appendix A.4. □

Let us now consider how full abstraction can be formulated for field calculus, through the following two predicates:

- $e_1 \simeq_{op} e_2$ for closed field calculus expressions e_i if and only if e_1 and e_2 evaluate to the same values in each firing of any possible operational evolution;
- $e_1 \simeq_{op} e_2$ for possibly non-closed field calculus expressions e_i iff for all contexts C such that $C[e_i]$ has observable type, $C[e_1]$ and $C[e_2]$ evaluate to the same values in each firing of any possible operational evolution.

Even though relation \simeq_{op} mirrors more closely the classical notions of full abstraction, it is not suitable to be used since the field calculus operational semantics includes checks for syntactic equality.

PROPOSITION 6.3 (SYNTACTIC EQUALITY). *Let e_1, e_2 be possibly non-closed field calculus expressions. Then $e_1 \simeq_{op} e_2$ if and only if $e_1 = e_2$ (syntactically).*

PROOF. See Appendix A.4. □

¹⁶We recall that $\mathcal{B}\llbracket b \rrbracket$ is based on the sensor information contained in events ϵ , while $\llbracket b \rrbracket_{\delta}^{\Theta, \sigma}$ is similarly based on the sensor state σ .

Proposition 6.3 implies that the only fully abstract semantics (in a classical sense) for a field calculus is the trivial semantics $\llbracket e \rrbracket = e$. Furthermore, it implies that common equivalences in classical scenarios do not hold in this case:

- It is not true that computational adequacy implies the left-to-right direction of full abstraction while the right-to-left direction is the hard one. Instead, the right-to-left direction is trivial (much unlike the PCF case [48]) and the left-to-right direction is false for any denotational semantics other than the trivial semantics.
- It is also not true that $=_{op}$ for closed expressions is equivalent to \simeq_{op} , as it would be in a classical scenario.

Altogether, these observations suggest that the best formulation of abstraction we can have for a field calculus is the following *computational abstraction* property:

$$\forall E. \mathcal{E} \llbracket e_1 \rrbracket^E = \mathcal{E} \llbracket e_2 \rrbracket^E \Leftrightarrow e_1 \simeq_{op} e_2,$$

where E is meant to include all the informations about the denotational environment and e_1, e_2 are closed expressions. The operational and denotational semantics hereby presented satisfy computational abstraction, *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 $c(\bar{\ell})$, $c'(\bar{\ell}')$ necessarily denote different objects.¹⁷

THEOREM 6.4 (COMPUTATIONAL ABSTRACTION). *Suppose that constructors for built-in local types are faithful. Then for every closed expressions e_1, e_2 , the following holds:*

$$\forall E. \mathcal{E} \llbracket e_1 \rrbracket^E = \mathcal{E} \llbracket e_2 \rrbracket^E \Leftrightarrow e_1 \simeq_{op} e_2,$$

where E includes all the informations about the denotational environment and $e_1 \simeq_{op} e_2$ if and only if e_1 and e_2 evaluate to the same values in each firing of any possible operational evolution.

PROOF. See Appendix A.4. □

6.3 Applications of Computational Abstraction

Thanks to the computational abstraction result, the denotational semantics can be used to formulate and prove conveniently intuitive facts about the calculus, which are then automatically true for practical implementations of HFC (which are built through the operational semantics). Few examples of such facts follows.

Alignment. Given any expression e of field type, $\mathcal{E} \llbracket e \rrbracket^E(\epsilon)$ is a neighbouring field with domain $E^-(\epsilon)$, provided that built-in functions with field return type respect this condition. This fact can be easily checked in the rule for `nbr`. In the case of function application, notice that e' has to be a value since it has type $(\bar{T}) \rightarrow F$. Thus $E(e', \epsilon) = E$ and the thesis follows by inductive hypothesis using `nbr` and built-in functions as base case.

Restriction. Given any expression e_0 executed in domain E and $V = \mathcal{E} \llbracket e_0 \rrbracket^E(\epsilon)$ for some $\epsilon \in E$, we say that $(\mathcal{E} \llbracket e_0 \rrbracket^E)^{-1}(V) = E(e_0, \epsilon)$ is a cluster. We say that function call has the restriction property to mean that well-typed expression $e_0(e_1, \dots, e_n)$ computes in isolation in each such cluster.

Namely, given e_0 and any of its clusters $E(e_0, \epsilon)$, let e_1, \dots, e_n and e'_1, \dots, e'_n be such that the denotation of e_i coincides with that of e'_i on $E(e_0, \epsilon)$, that is $\mathcal{E} \llbracket e_i \rrbracket^E|_{E(e_0, \epsilon)} = \mathcal{E} \llbracket e'_i \rrbracket^E|_{E(e_0, \epsilon)}$. Then $\mathcal{E} \llbracket e_0(e_1, \dots, e_n) \rrbracket^E(\epsilon) = \mathcal{E} \llbracket e_0(e'_1, \dots, e'_n) \rrbracket^E(\epsilon)$ for any $\epsilon \in E(e_0, \epsilon)$.

¹⁷This property fails, for example, if we include constructors `Succ`, `Pred` for type `num`. However, it can hold for example if we assume to have a distinguished constructor n for every integer n .

This means that computation of e_0 inside cluster $E(e_0, \epsilon)$ 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 2 and implies the analogous property for the first-order calculus with conditionals (since the conditional expression can be simulated by function application).

rep-expressions. Consider an expression $e = \text{rep}(e_1)\{(x) \Rightarrow e_2\}$ and suppose that ϵ is a “source” event in E , that is, there exists no ϵ' in E such that $\text{neigh}(\epsilon, \epsilon')$. Then $\mathcal{E}[\![e]\!](\epsilon) = \mathcal{E}[\![(x) \Rightarrow e_2](e_1)\!](\epsilon)$.

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

7 RELATED WORK

The work on field calculus presented in this article builds on a sizable body of prior work. We begin with a general review of works on the programming of aggregates, and following this, we compare this article with previous work by the same authors.

7.1 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 to adapt to unexpected contingencies such as device failures, loss of messages, changes to inputs, modification of network topology, and so on. 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 References [8, 54], 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:

- *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) [15, 16, 43], shared-space abstractions by which multiple processes can interact in a decoupled way [13, 55], and attribute-based models declaratively specifying the target of communication so as to dynamically create ensembles [26].
- *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 [39] allows one to program tuples with reaction and diffusion rules, while in the SAPERE approach [58] such rules are embedded in space and apply semantically, and the $\sigma\tau$ -Linda model [57] manipulates tuples over space and time. Other examples include MPI [41], which declaratively expresses topologies of processes in supercomputing applications,

- NetLogo [51], which provides abstract means to interact with neighbours following the cellular automata style, and Hood [59], 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 [44] allows the programmer to imperatively specify geometric folds that are compiled into processes identifying regions of space, Growing Point Language [19] provides means to describe topologies in terms of a “botanical” metaphor with growing points and tropisms, ASCAPE [35] supports agent communication by means of topological abstractions and a rule language, and the catalogue of self-organisation patterns in Reference [28] 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 [38] viewing a wireless sensor network as a database, Regiment [45] using a functional language to be compiled into protocols of device-to-device interaction, and the agent communication language KQML [29].
 - *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 [7] is a Lisp-like functional language and simulator for programming wireless sensor networks with the notion of computational fields, and MGS [32] 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 Reference [9], 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.

7.2 Aggregate Computing with Fields

At the core of the approach presented in this article 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 [9] up to a point in which the focus on individual agent behaviour completely vanishes. This compositional stacking of increasingly complex distributed algorithms is at the core of the *aggregate computing* paradigm [9] and can be taken even further by proving that the “building block” algorithms satisfy certain properties preserved by functional compositions, such as self-stabilisation [53] or consistency with a continuum model [11], thus implying the same properties hold for applications composed using those building blocks [9].

The calculus that we present in this article is a higher-order extension of the work in Reference [23] to include embedded first-class functions, with the primary goal of allowing field computations to handle functions just like any other value while clarifying syntax and semantics of the field-based computational model. 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 interpreted as 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.

The last feature is critical: In considering fields of function values, we took 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, allowing different pieces of code to be executed without unwanted interactions. 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 insights of this article, 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.

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 7. 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 Reference [56], that for the first time smoothly combines self-organisation 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 [50], 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 References [10] and [9]. We aim to apply our approach to support various application needs for dynamic management of distributed processes [6], 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 References [53] and [22]. Finally, these foundations can be applied in developing APIs enabling the simple construction of complex distributed applications, building on the work in References [2–4, 9, 53] to define a layered library of self-organisation patterns, and applying these APIs to support a wide range of practical distributed applications.

APPENDIXES

A PROOFS OF THE MAIN RESULTS

Supplementary materials, including proofs of the main results and a pervasive computing example, are available in the online version of this article.

A.1 Refined Monomorphic Typing (Proof of Theorem 5.1)

RESTATEMENT OF THEOREM 5.1 (SYSTEM \vdash' REFINES SYSTEM \vdash). *If $S_0 \vdash' P : T$, then $S_0 \vdash P : T$.*

PROOF. Just observe that every derivation of $S; \mathcal{M} \vdash' e : T$ is also a derivation of $S; \mathcal{M} \vdash e : T$, since each rule of \vdash' is a restriction of the corresponding rule of \vdash and similarly for user-defined function declarations and programs. \square

A.2 Hindley-Milner Typing vs. Monomorphic Typing (Proof of Theorem 5.3)

We say that a monomorphic type-set environment S is an *instantiation* of a type-scheme environment \mathcal{D} if and only if $\text{dom}(S) = \text{dom}(\mathcal{D})$ and for each g in $\text{dom}(S)$, $S(g) \subseteq \text{MS}(\mathcal{D}(g))$. Furthermore, we say that S is a *full instantiation* of \mathcal{D} if additionally $S(g) = \text{MS}(\mathcal{D}(g))$ for each g in their (common) domain. Finally, we say that a monomorphic type T' is an *instantiation* of a (possibly) parametric type T if $T' = T[\bar{l} := \bar{T}, \bar{l} := \bar{L}, \bar{r} := \bar{R}, \bar{s} := \bar{S}]$ and that a monomorphic type environment \mathcal{M} is an *instantiation* of a type environment \mathcal{A} if and only if $\text{dom}(\mathcal{M}) = \text{dom}(\mathcal{A})$ and for each x in $\text{dom}(\mathcal{M})$, $\mathcal{M}(x) \in \text{MS}(\mathcal{A}(x))$.

LEMMA A.1 (FROM MONOMORPHIC TYPING \vdash' TO HINDLEY-MILNER TYPING \vdash). *Assume that S_0 is a full instantiation of \mathcal{D}_0 and S is an instantiation of \mathcal{D} with $S_0 \subseteq S$, $\mathcal{D}_0 \subseteq \mathcal{D}$. Then:*

- (1) $S; \mathcal{M} \vdash' e : T$ implies $\mathcal{D}; \mathcal{M} \vdash e : T$ for any monomorphic T ;
- (2) $S \vdash' F : Lset$ implies $\mathcal{D} \vdash F : LS$, for some LS such that $Lset \subseteq \text{MS}(LS)$;
- (3) $S_0 \vdash' P : T$ implies $\mathcal{D}_0 \vdash P : T$.

PROOF. We prove (1)–(3) by simultaneous induction on the syntax of the involved expressions.

- (1) $e = c(\bar{\ell})$ is a data expression. Since $S_0(c) = \text{MS}(\mathcal{D}_0(c))$, Rules [M-DAT'] and [T-DAT] assign the same monomorphic types $(\bar{S}) \rightarrow S$ to c (left premise). By inductive hypothesis, they also assign the same monomorphic types to $\bar{\ell}$ (right premise), concluding the proof. $e = g$ is a user-defined function. Rule [M-N-FUN'] assigns to g every type in $S(g)$, while Rule [T-N-FUN] assigns to g every type in $\text{MS}(\mathcal{D}(g)) \supseteq S(g)$, concluding the proof. e is not a data expression nor a user-defined function. The thesis follows since Rules [M-VAR'], [M-A-FUN'], [M-APP'], [M-REP'], [M-NBR'] are syntax-directed and identical to rules [T-VAR], [T-A-FUN], [T-APP], [T-REP], [T-NBR].
- (2) Assume that F is $\text{def}d(\bar{x})\{e\}$ and $S \vdash' F : Lset$. Then by Rule [M-FUNCTION'], we have that $S, d : (\bar{T}) \rightarrow R; \bar{x} : \bar{T} \vdash' e : R$ for each $(\bar{T}) \rightarrow R \in Lset$. It follows that

$\mathcal{D}, d : (\bar{T}) \rightarrow R; \bar{x} : \bar{T} \vdash e : R$ by inductive hypothesis (point 1), hence $\mathcal{D} \vdash F : (\bar{T}) \rightarrow R$ by Rule [T-FUNCTION]. Let LS be the principal type assigned to F : then every other type assignable to F (including every $(\bar{T}) \rightarrow R \in Lset$) is in $\mathbf{MS}(LS)$, concluding the proof.

- (3) Let P be $F_1 \cdots F_n e$. Since Rule [M-PROGRAM] is identical to Rule [T-PROGRAM], we can use the inductive hypothesis (point 2) to prove by induction on $i \leq n$ that $\mathbf{dom}(S_i) = \mathbf{dom}(\mathcal{D}_i)$ and $\forall g \in \mathbf{dom}(S_i). S_i(g) \subseteq \mathbf{MS}(\mathcal{D}_i(g))$. Thus by inductive hypothesis (point 1), $S_n; \emptyset \vdash' e : T$ implies that $\mathcal{D}_n; \emptyset \vdash e : T$ concluding the proof. \square

LEMMA A.2 (FROM HINDLEY-MILNER TYPING \vdash TO MONOMORPHIC TYPING \vdash'). Assume that S_0 is a full instantiation of \mathcal{D}_0 and $\mathcal{D}_0 \subseteq \mathcal{D}$. Then:

- (1) $\mathcal{D}; \mathcal{A} \vdash e : T$ implies that for each T' instantiation of T , there exist S, \mathcal{M} instantiations of \mathcal{D}, \mathcal{A} such that $S_0 \subseteq S$ and $S; \mathcal{M} \vdash e : T'$.
- (2) $\mathcal{D} \vdash F : LS$ implies that for each finite $Lset \subseteq \mathbf{MS}(LS)$, there exists S instantiation of \mathcal{D} such that $S_0 \subseteq S$ and $S \vdash' F : Lset$.
- (3) $\mathcal{D}_0 \vdash P : T$ implies that for each T' instantiation of T , $S_0 \vdash' P : T'$.

PROOF. We prove (1)–(3) by simultaneous induction on the syntax of the involved expressions.

- (1) Let $T' = T[\bar{\ell} := \bar{T}, \bar{\ell} := \bar{L}, \bar{r} := \bar{R}, \bar{s} := \bar{S}]$ and \mathcal{A}' be obtained from \mathcal{A} through the same substitutions, so that $\mathcal{D}; \mathcal{A}' \vdash e : T'$.

$e = c(\bar{\ell})$ is a data expression. Let $(\bar{S}) \rightarrow S$ be (an instantiation of) the type assigned to c by Rule [T-DAT] (left premise) in $\mathcal{D}; \mathcal{A}' \vdash e : T'$. Since S_0 is a full instantiation of \mathcal{D}_0 , $(\bar{S}) \rightarrow S \in S_0(c)$ and $S_0; \emptyset \vdash' \bar{\ell} : \bar{S}$. By Rule [M-DAT], the thesis holds for any S, \mathcal{M} instantiations of $\mathcal{D}, \mathcal{A}'$ such that $S_0 \subseteq S$.

$e = g$ is a user-defined function. Let S be any instantiation of \mathcal{D} such that $T' \in S(g)$ and $S_0 \subseteq S$. Then the thesis follows by Rule [M-N-FUN].

e is not a data expression nor a user-defined function. By inductive hypothesis, let S_i, \mathcal{M}_i for $i \leq m$ be instantiations of \mathcal{D}, \mathcal{A} that translate the i th premise of the last rule to \vdash' . For each i , let $\mathcal{M}'_i \subseteq \mathcal{M}_i$ be collecting the type assumptions actually used in the i th premise. Since all the rules are syntax-directed, the \mathcal{M}'_i are distinct and any $\mathcal{M} \supseteq \bigcup_{i \leq m} \mathcal{M}_i$ instantiation of \mathcal{A} translates each premise to \vdash' . Let S be an instantiation of \mathcal{D} such that $S(g) \supseteq \bigcup_{i \leq m} S_i(g)$ for all $g \in \mathbf{dom}(\mathcal{D})$. Then, the thesis holds for S, \mathcal{M} since Rules [M-VAR], [M-A-FUN], [M-APP], [M-REP], [M-NBR] are identical to rules [T-VAR], [T-A-FUN], [T-APP], [T-REP], [T-NBR].

- (2) Assume that F is $\text{def } d(\bar{x})\{e\}$ and $Lset = \{\bar{T}^{(1)} \rightarrow R^{(1)}, \dots, \bar{T}^{(m)} \rightarrow R^{(m)}\}$. By inductive hypothesis (point 1), there are instantiations S_i of \mathcal{D} with $S_0 \subseteq S_i$ such that:

$$S_i, d : \{\bar{T}^{(i)} \rightarrow R^{(i)}\}; \bar{x} : \bar{T}^{(i)} \vdash e : R^{(i)}.$$

Then the thesis holds for any S instantiation of \mathcal{D} such that $S(g) \supseteq \bigcup_{i \leq m} S_i(g)$ for all g in $\mathbf{dom}(\mathcal{D})$.

- (3) Let P be $F_1 \cdots F_n e$ with $F_i = \text{def } d_i(_) \{ _ \}$, and let \mathcal{D}_i for $i = 0 \dots n$ be the local type-scheme environments appearing in Rule [T-PROGRAM]. By inductive hypothesis (point 1), let S'_n be an instantiation of \mathcal{D}_n containing S_0 such that $S'_n; \emptyset \vdash' e : T'$. By inductive hypothesis (point 2), let S'_i for $i = n-1 \dots 0$ be an instantiation of \mathcal{D}_i containing S_0 such that $S'_i \vdash' F_{i+1} : Lset_{i+1}$ where $Lset_{i+1} = \bigcup_{j=i+1 \dots n} S'_j(d_{i+1})$. Since the only instantiation of \mathcal{D}_0 containing S_0 is S_0 , it follows that $S'_0 = S_0$. Let S_i for $i = 1 \dots n$ be $S_0 \cup \{d_j \mapsto Lset_j : j = 1 \dots i\}$. Since S'_i is a point-wise subset of S_i , $S_i \vdash' F_{i+1} : Lset_{i+1}$ as well. Thus, the thesis follows by Rule [T-PROGRAM]. \square

RESTATEMENT OF THEOREM 5.3 (HINDLEY-MILNER TYPING \vdash VS. MONOMORPHIC TYPING \vdash'). Assume that \mathcal{S}_0 and \mathcal{D}_0 have the same domain and for each g in their domain it holds that $\mathcal{S}_0(g) = \mathbf{MS}(\mathcal{D}_0(g))$. Then, for any monomorphic type T , it holds that $\mathcal{D}_0 \vdash P : T$ if and only if $\mathcal{S}_0 \vdash' P : T$.

PROOF. Straightforward by Lemma A.1 and Lemma A.2. \square

A.3 Device Computation Type Preservation and Domain Alignment (Proof of Theorem 6.1)

RESTATEMENT OF THEOREM 6.1 (TYPE PRESERVATION AND DOMAIN ALIGNMENT). Let $\mathcal{A} = \bar{x} : \bar{T}$, $\mathcal{D}; \emptyset \vdash \bar{v} : \bar{T}$, so that $\text{length}(\bar{v}) = \text{length}(\bar{x})$. If $\mathcal{D}; \mathcal{A} \vdash e : T$, $\Theta \in \text{WFVTE}(\mathcal{D}, \mathcal{A}, e)$ and $\delta; \Theta; \sigma \vdash e[\bar{x} := \bar{v}] \Downarrow \theta$, then:

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

Observe that the typing rules (in Figure 10) and the evaluation rules (in Figure 7) are syntax directed. Then the proof can be carried out by induction on the syntax of expressions, while using the following standard lemmas.

LEMMA A.3 (SUBSTITUTION). Let $\mathcal{A} = \bar{x} : \bar{T}$, $\mathcal{B}; \emptyset \vdash \bar{v} : \bar{T}$. If $\mathcal{D}; \mathcal{A} \vdash e : T$, then $\mathcal{D}; \emptyset \vdash e[\bar{x} := \bar{v}] : T$.

PROOF. Straightforward by induction on application of the typing rules for expressions in Figure 10. \square

LEMMA A.4 (WEAKENING). Let $\mathcal{D}' \supseteq \mathcal{D}$, $\mathcal{A}' \supseteq \mathcal{A}$ be such that $\text{dom}(\mathcal{D}') \cap \text{dom}(\mathcal{A}') = \emptyset$. If $\mathcal{D}; \mathcal{A} \vdash e : T$, then $\mathcal{D}'; \mathcal{A}' \vdash e : T$.

PROOF. Straightforward by induction on application of the typing rules for expressions in Figure 10. \square

PROOF OF THEOREM 6.1. We proceed proving points (1)–(3) by simultaneous induction on the syntax of expression e (given in Figure 1).

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

If e has field type, then we also need to check that $\text{dom}(\rho(\theta)) = \text{dom}(\Theta) \cup \{\delta\}$. In this case, e_{n+1} is of type $(\bar{T}) \rightarrow F$, which is not a return type: thus, e_{n+1} cannot be of the form rep , nbr , $c(\dots)$ or $e'(\bar{e}')$, since all these constructs are only given return types. It follows that e_{n+1} is in fact equal to a value f , hence no alignment with neighbours is required, directly concluding the proof whenever f is not a built-in function. If instead f is a built-in

function, then the thesis follows from the assumptions on the interpretation function $\langle \mathbf{b} \rangle_{\delta}^{\Theta, \sigma}$ (i.e., that fields returned have domain $\mathbf{dom}(\Theta)$).

- $e = \text{nbr}\{e_1\}$: In this case, rule $[\text{E-NBR}]$ 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 e , and ϕ has type $T = \text{field}(T_1)$ where $e_1 : T_1$. Furthermore, $\mathbf{dom}(\phi) = \mathbf{dom}(\Theta) \cup \{\delta\}$ as required.
- $e = \text{rep}(e_1)\{(x) \Rightarrow e_2\}$: In this case, rule $[\text{E-REP}]$ applies, thus $\theta = \ell_2\langle\theta_1, \theta_2\rangle$ is well-formed for e (where ℓ_i, θ_i follows the notation in Figure 7, rule $[\text{E-REP}]$). By induction hypothesis, $\ell_1 = \rho(\theta_1)$ has the same type as e_1 , which is T . If $\delta \notin \mathbf{dom}(\Theta)$, then $\ell_0 = \ell_1$ also has type T . Otherwise, since Θ is well-formed for e , $\pi_2(\Theta)$ is well-formed for e_2 with the additional assumption that $x : T$, and by induction hypothesis $\ell_0 = \rho(\pi_2(\Theta))(\delta)$ has the same type as e_2 , which is also T . Then by the Substitution Lemma, $e_2[x := \ell_0]$ also has type T hence by induction hypothesis the same does $\ell_2 = \rho(\theta_2)$, concluding the proof. \square

A.4 Computational Adequacy and Abstraction (Proof of Theorem 6.2, Proposition 6.3 and 6.4)

RESTATEMENT OF THEOREM 6.2 (COMPUTATIONAL ADEQUACY). *Assume that the denotational environment is coherent with the operational evolution of the network and e is well typed (that is, $\mathcal{D}; \emptyset \vdash e : T$).*

Then $\mathcal{E}\llbracket e \rrbracket^E(\epsilon) = \mathcal{E}\llbracket v_\epsilon \rrbracket^E(\epsilon)$ for each ϵ in E , where $v_\epsilon = \rho(\theta_\epsilon)$ is the operational outcome of fire ϵ .

To carry on the induction on the structure of e , we shall prove the following strengthened version instead.

LEMMA A.5. *Assume that the denotational environment is coherent with the operational evolution of the network and e is a well-typed expression with free variables \bar{x} (that is, $\mathcal{D}; \bar{x} : \bar{T} \vdash e : T$). Let $X = \bar{x} \mapsto \bar{\Phi}$ and \bar{u}^ϵ be such that $\Phi_i(\epsilon) = \mathcal{E}\llbracket u_i^\epsilon \rrbracket(\epsilon)$ for all i and events ϵ .*

Then $\mathcal{E}\llbracket e \rrbracket_X(\epsilon) = \mathcal{E}\llbracket v^\epsilon \rrbracket(\epsilon)$ for each ϵ in E , where $v^\epsilon = \rho(\theta_\epsilon)$ is the operational outcome of fire ϵ evaluating $e[\bar{x} := \bar{u}^\epsilon]$.

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

- $e = x_i$: In this case, $e[\bar{x} := \bar{u}^\epsilon] = u_i^\epsilon = v^\epsilon$ and by hypothesis $\Phi_i(\epsilon) = \mathcal{E}\llbracket u_i^\epsilon \rrbracket(\epsilon) = \mathcal{E}\llbracket v^\epsilon \rrbracket(\epsilon)$.
- $e = c(\bar{\ell}) \mid \phi \mid b \mid d$: Since e is already a value, $v^\epsilon = e$ and the thesis follows.
- $e = (\bar{x}) \stackrel{\tau}{\Rightarrow} e$: Since both the operational and denotational semantics compare anonymous function values through their name τ alone, the substitutions X do not affect the interpretations of e (which is already a value), and the thesis follows.
- $e = \text{nbr}\{e_1\}$: By inductive hypothesis, $\mathcal{E}\llbracket e_1 \rrbracket_X(\epsilon) = \mathcal{E}\llbracket v_1^\epsilon \rrbracket(\epsilon)$ for each ϵ in E , where $v_1^\epsilon = \rho(\theta_1^\epsilon)$ is the (operational) outcome of $e_1[\bar{x} := \bar{u}^\epsilon]$ in fire ϵ . By rule $[\text{E-NBR}]$, we have that

$$v^\epsilon = \{\delta_{\epsilon'} \mapsto v_1^{\epsilon'} : (\epsilon' = \epsilon) \vee (\text{neigh}(\epsilon, \epsilon') \wedge \delta_{\epsilon'} \neq \delta_\epsilon)\}$$

and

$$\mathcal{E}\llbracket v^\epsilon \rrbracket(\epsilon) = \{\delta_{\epsilon'} \mapsto \mathcal{E}\llbracket v_1^{\epsilon'} \rrbracket(\epsilon) : (\epsilon' = \epsilon) \vee (\text{neigh}(\epsilon, \epsilon') \wedge \delta_{\epsilon'} \neq \delta_\epsilon)\}.$$

On the other hand, $\mathcal{E}[\![e]\!]_X(\epsilon)$ is

$$\begin{aligned}\mathcal{E}[\![\text{nbr}\{e_1\}]\!]_X(\epsilon) &= \lambda\delta \in E^-(\epsilon). \mathcal{E}[\![e_1]\!]_X(\epsilon^\delta) \\ &= \lambda\delta \in E^-(\epsilon). \mathcal{E}[\![v_1^{\epsilon^\delta}]\!]_X(\epsilon^\delta) \\ &= \{\delta \mapsto \mathcal{E}[\![v_1^{\epsilon^\delta}]\!]_X(\epsilon) : (\epsilon^\delta = \epsilon) \vee (\text{neigh}(\epsilon, \epsilon^\delta) \wedge \delta \neq \delta_\epsilon)\} = \mathcal{E}[\![v^\epsilon]\!]_X(\epsilon).\end{aligned}$$

- $e = \text{rep}(e_1)\{(y) \Rightarrow e_2\}$: Recall that by rule [E-REP], v^ϵ is the evaluation of $e_2[\bar{x} := \bar{u}^\epsilon, y := v_1^{\epsilon^-}]$ if ϵ^- exists, $e_2[\bar{x} := \bar{u}^\epsilon, y := v_1^\epsilon]$ 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 $\mathcal{E}[\![v^\epsilon]\!]_X(\epsilon) = \mathcal{R}[\![e]\!]_{n+1}(\epsilon)$. The thesis will then follow for n greater than the number of events.

If $n = 0$, then by induction hypothesis on both e_1 and $e_2[\bar{x} := \bar{u}^\epsilon, y := v_1^\epsilon]$:

$$\mathcal{E}[\![v^\epsilon]\!]_X(\epsilon) = \mathcal{E}[\![e_2]\!]_{X \cup y \mapsto \mathcal{E}[\![e_1]\!]_X}(\epsilon) = \mathcal{R}[\![e]\!]_1(\epsilon).$$

If $n > 0$, then v^{ϵ^-} has at most $n - 1$ predecessors thus we can apply the inductive hypothesis on $n - 1$ and $e_2[\bar{x} := \bar{u}^\epsilon, y := v_1^{\epsilon^-}]$ to obtain:

$$\mathcal{E}[\![v^\epsilon]\!]_X(\epsilon) = \mathcal{E}[\![e_2]\!]_{X \cup y \mapsto \text{shift}(\mathcal{R}[\![e]\!]_n, \mathcal{R}[\![e]\!]_0)}(\epsilon) = \mathcal{R}[\![e]\!]_{n+1}(\epsilon).$$

- $e = e_{n+1}(\bar{e})$: Suppose that \bar{e} evaluates to \bar{v}^ϵ and e_{n+1} to f^ϵ in event ϵ . Notice that by induction hypothesis, $\mathcal{E}[\![\bar{v}^\epsilon]\!]_X(\epsilon) = \mathcal{E}[\![\bar{e}]\!]_X(\epsilon)$ and $\mathcal{E}[\![f^\epsilon]\!]_X(\epsilon) = \mathcal{E}[\![e_{n+1}]\!]_X(\epsilon)$.

If f^ϵ is a built-in function b , then by rule [E-B-APP] $b(\bar{v}^\epsilon)$ (hence $e[\bar{x} := \bar{u}^\epsilon]$) evaluates in ϵ to the result v^ϵ of $\langle b \rangle_{\delta_\epsilon}^{\pi^b(\Theta_\epsilon), \sigma_\epsilon}(\bar{v}^\epsilon)$, calculated in the restricted environment $\Theta'_\epsilon = \pi^b(\Theta_\epsilon)$.

The value-tree environments $\{\Theta'_{\epsilon'} : f^{\epsilon'} = b\}$ together define an operational environment Env_b consisting only of those devices and events that agree on the evaluation of e_{n+1} . In fact, such restricted environment is coherent with the restricted denotational environment $E(e_{n+1}, \epsilon)$:

$$\begin{aligned}E(e_{n+1}, \epsilon) &= \{\epsilon' : \mathcal{E}[\![e_{n+1}]\!]_X(\epsilon') = \mathcal{E}[\![e_{n+1}]\!]_X(\epsilon)\} \\ &= \{\epsilon' : \mathcal{E}[\![f^{\epsilon'}]\!]_X(\epsilon') = \mathcal{E}[\![b]\!]_X(\epsilon)\} \\ &= \{\epsilon' : \mathcal{E}[\![f^{\epsilon'}]\!]_X = \mathcal{E}[\![b]\!]_X\} = \{\epsilon' : f^{\epsilon'} = b\}\end{aligned}$$

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). The thesis then follows from coherence of $\mathcal{B}[\![b]\!]$ with $\langle b \rangle_{\delta}^{\Theta, \sigma}$, together with the induction hypothesis on e_1, \dots, e_n .

If f^ϵ is an anonymous function, then by rule [E-D-APP] $f^\epsilon(\bar{v}^\epsilon)$ (hence $e[\bar{x} := \bar{u}^\epsilon]$) evaluates in ϵ to the result v^ϵ of $\text{body}(f^\epsilon)[\text{args}(f^\epsilon) := \bar{v}^\epsilon]$ as calculated in $\Theta'_\epsilon = \pi^{f^\epsilon}(\Theta_\epsilon)$. As for built-in functions, the operational environment $\text{Env}_{f^\epsilon} = \{\Theta'_{\epsilon'} : f^{\epsilon'} = f^\epsilon\}$ is coherent with the restricted denotational environment $E(e_{n+1}, \epsilon)$. Thus we can apply the induction hypothesis on \bar{e} in E and on $\text{body}(f^\epsilon)[\text{args}(f^\epsilon) := \bar{v}^\epsilon]$ in $E(e_{n+1}, \epsilon)$ to obtain:

$$\mathcal{E}[\![v^\epsilon]\!]_X(\epsilon) = \mathcal{E}[\![\text{body}(f^\epsilon)]\!]_{\text{args}(f^\epsilon) \mapsto \mathcal{E}[\![\bar{e}]\!]_X|_{E(e_{n+1}, \epsilon)}}^{E(e_{n+1}, \epsilon)}(\epsilon)$$

On the other hand,

$$\begin{aligned}
 \mathcal{E}[\![e]\!]_X(\epsilon) &= \text{snd} \left(\mathcal{E}[\![e_{n+1}]\!]_X(\epsilon) \right) \left(\mathcal{E}[\![\bar{e}]\!]_X|_{\mathbb{E}(e_{n+1}, \epsilon)}(\epsilon) \right) \\
 &= \text{snd} \left(\mathcal{E}[\![f^\epsilon]\!]_X(\epsilon) \right) \left(\mathcal{E}[\![\bar{e}]\!]_X|_{\mathbb{E}(e_{n+1}, \epsilon)}(\epsilon) \right) \\
 &= \left(\lambda \bar{\Phi}. \mathcal{E}[\![\text{body}(f^\epsilon)]\!]_{\text{args}(f^\epsilon) \mapsto \bar{\Phi}}^{\text{dom}(\bar{\Phi})} \right) \left(\mathcal{E}[\![\bar{e}]\!]_X|_{\mathbb{E}(e_{n+1}, \epsilon)}(\epsilon) \right) \\
 &= \mathcal{E}[\![\text{body}(f^\epsilon)]\!]_{\text{args}(f^\epsilon) \mapsto \mathcal{E}[\![\bar{e}]\!]_X^{\mathbb{E}}|_{\mathbb{E}(e_{n+1}, \epsilon)}}^{\mathbb{E}(e_{n+1}, \epsilon)}(\epsilon) = \mathcal{E}[\![v^\epsilon]\!]_X(\epsilon)
 \end{aligned}$$

completing the proof in this case.

If f^ϵ is a user-defined function, then we prove by further induction on n that $\mathcal{E}[\![v^\epsilon]\!]_X(\epsilon)$ is equal to $V_n = \mathcal{E}[\![\text{body}(f^\epsilon)]\!]_{\text{args}(f^\epsilon) \mapsto \mathcal{E}[\![\bar{e}]\!]_X^{\mathbb{E}}|_{\mathbb{E}(e_{n+1}, \epsilon)}}^{\mathbb{E}(e_{n+1}, \epsilon)}(\epsilon)$ 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 e as for anonymous functions).

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

$$V_0 = \mathcal{E}[\![\text{body}(f^\epsilon)]\!]_{\text{args}(f^\epsilon) \mapsto \mathcal{E}[\![\bar{e}]\!]_X^{\mathbb{E}}|_{\mathbb{E}(e_{n+1}, \epsilon)}}^{\mathbb{E}(e_{n+1}, \epsilon)}(\epsilon) = \mathcal{E}[\![v^\epsilon]\!]_X(\epsilon)$$

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

RESTATEMENT OF PROPOSITION 6.3 (SYNTACTIC EQUALITY). *Let e_1, e_2 be possibly non-closed field calculus expressions. Then $e_1 =_{op} e_2$ if and only if $e_1 = e_2$ (syntactically).*

PROOF. The right-to-left direction is trivial. Assume then that $e_1 \neq e_2$ and let \bar{x} be the list of free variables $\mathbf{FV}(e_1) \cup \mathbf{FV}(e_2)$. Consider the following context of observable type num :¹⁸

```

def countneigh(v) { ;; has type:  $\forall t. (t) \rightarrow \text{num}$ 
  sum-hood( nbr{1} )
}
C[ · ] := mux( uid() = 0, ;; has type: num
  ( ) => countneigh(  $\bar{x}$  ) => countneigh( $e_1$ ) ),
  ( ) => countneigh(  $\bar{x}$  ) => countneigh([ · ] )
)( )

```

Notice that function `countneigh` ignores its argument while always returning the number of aligned neighbours. In any operational evolution where device $\delta = 0$ is connected to other devices:

- in $C[e_1]$, the two branches of the `mux` operator are syntactically identical hence all devices are aligned and the total number of neighbours is computed in each device;
- in $C[e_2]$, the two branches are different hence device $\delta = 0$ does not have aligned neighbours, computing 0, which is less than the total number of neighbours.

It follows that $e_1 \neq_{op} e_2$ concluding the proof. \square

RESTATEMENT OF THEOREM 6.4 (COMPUTATIONAL ABSTRACTION). *Suppose that constructors for built-in local types are faithful. Then for every closed expressions e_1, e_2 , the following holds:*

$$\forall E. \mathcal{E}[\![e_1]\!]^E = \mathcal{E}[\![e_2]\!]^E \iff e_1 \simeq_{op} e_2.$$

¹⁸The innermost occurrence of `countneigh` allows for sound typing in case e_1, e_2 are not of a return type, so that they cannot be directly closed.

where E includes all the informations about the denotational environment and $e_1 \simeq_{op} e_2$ if and only if e_1 and e_2 evaluate to the same values in each firing of any possible operational evolution.

PROOF. First notice that given any two values v_1, v_2 of the same type $\mathcal{E}[\![v_1]\!]^E = \mathcal{E}[\![v_2]\!]^E$ if and only if $v_1 = v_2$. If v_1, v_2 are functions, then it holds since the denotation includes the function tag (i.e., the syntactic expression itself). If v_1, v_2 are constructor expressions, then it holds by hypothesis. If v_1, v_2 are neighbouring field values, then it holds since $\mathcal{E}[\![\bar{\delta} \mapsto \bar{\ell}]\!]^E = \lambda \epsilon. \bar{\delta} \mapsto \mathcal{E}[\![\bar{\ell}]\!]^E(\epsilon)$ and we already proved the equivalence for local values. From this equivalence and computational adequacy we can prove the computational abstraction property as follows.

Suppose that e_1 and e_2 have the same denotation in every denotational environment and fix an operational environment Env , in which e_1, e_2 evaluate to $v_\epsilon^1, v_\epsilon^2$ in fire ϵ . Then given a denotational environment E that is coherent with Env , we have that $\mathcal{E}[\![e_1]\!]^E = \mathcal{E}[\![e_2]\!]^E$. By adequacy, it follows that $\mathcal{E}[\![v_\epsilon^1]\!]^E(\epsilon) = \mathcal{E}[\![v_\epsilon^2]\!]^E(\epsilon)$ hence $v_\epsilon^1 = v_\epsilon^2$ for all ϵ . Thus e_1 and e_2 evaluate to the same value in every fire ϵ and the left-to-right part of the proof is concluded.

For the converse implication, suppose that there exists a denotational environment E such that e_1 and e_2 have denotations that differ in ϵ . Let Env be an operational environment coherent with E , in which e_1, e_2 evaluate to v_1, v_2 in fire ϵ . By adequacy,

$$\mathcal{E}[\![v_1]\!]^E(\epsilon) = \mathcal{E}[\![e_1]\!]^E(\epsilon) \neq \mathcal{E}[\![e_2]\!]^E(\epsilon) = \mathcal{E}[\![v_2]\!]^E(\epsilon)$$

hence $v_1 \neq v_2$ concluding the proof. \square

B 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., 30m). 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.

B.1 Virtual Machine Implementation

The complete code for our example is listed in Figure 12. We use the following naming conventions for built-ins: functions $sns\text{-}*$ embed sensors that return a value perceived from the environment (e.g., $sns\text{-}injection\text{-}point$ returns a Boolean indicating whether a device’s user wants to inject a function); functions $*\text{-}hood$ yield a local value ℓ obtained by aggregating over the neighbouring field value ϕ in the input (e.g., $sum\text{-}hood$ sums all values in each neighbourhood); functions $*\text{-}hood\text{-}$ 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.

The first two functions in Figure 12 implement frequently used self-organisation mechanisms. As already discussed, function $distance\text{-}to$, also known as *gradient* [18, 37], 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


```

;; 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 the most recent value of 'v' in the nearest 'source' device
def gradcast (source, v) { ;; has type: ∀ s. (bool, s) → s
  snd(
    rep (Pair(infinity, v)) { (t) =>
      mux(source, Pair(0, v), min-hood+(
        Pair[f,f](+[f,f](nbr-range(), nbr{fst(t)}), nbr{snd(t)}) )
    )
  )
}

;; Evaluate a function field, running 'g' from 'source' within 'range' meters, and 'no-op' elsewhere
def deploy (range, source, g, no-op) { ;; has type: ∀ s. (num, bool, () → s, () → s) → s
  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)) }
}

```

Fig. 12. Virtual machine code (top) and application-specific code (bottom).

desired field of distances, regardless of any transient perturbations or faults [36]. 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 12 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 $() \Rightarrow \emptyset$ 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., [14, 31, 39])—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 6.3), 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.

B.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: $() \rightarrow \text{num}$):

```
() => 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 7.2, 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 [53], so its output is passed through a simple low-pass filter, the function low-pass, also defined in Figure 12(bottom), to smooth its output and improve the quality of estimate.

Figure 13(a) shows a simulation of a docent and 250 patrons in a large $100 \times 30\text{m}$ 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 1m/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 10m —hence, hop-by-hop communication is needed for longer range interaction. The simulation was implemented using the ALCHEMIST [49] simulation framework and the Protelis [50] incarnation of field calculus, updated to the extended version of the calculus presented in this article.

In this simulation, at time 10s , the docent injects the patron-counting function with a range of 25m , and at time 70s removes it. Figure 13(a) shows two snapshots of the simulation, at times 11s (top) and 35s (bottom), while Figure 13(b) 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 field calculus and its safety properties allow complex distributed applications to be safely and elegantly implemented with compact code, including higher-order operations like process management and runtime deployment of applications in an open environment.

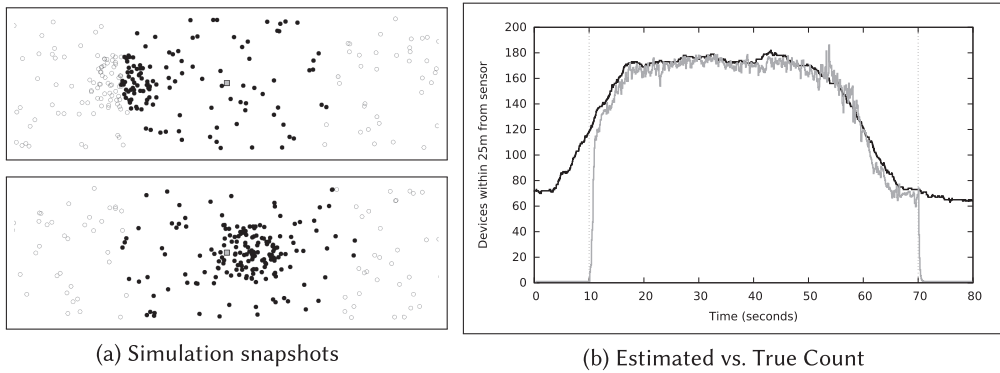


Fig. 13. (a) Two snapshots of museum simulation: patrons (grey) are counted (black) within 25m of the docent (green). (b) Estimated number of nearby patrons (grey) vs. actual number (black) in the simulation, during the period where the docent is running the patron counting function (between two dashed lines).

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for many insightful comments and suggestions.

REFERENCES

- [1] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. 2018. Space-time universality of field calculus. In *Proceedings of the Coordination Models and Languages (COORDINATION'18)*, Lecture Notes in Computer Science, Vol. 10852. Springer, 1–20. DOI: https://doi.org/10.1007/978-3-319-92408-3_1
- [2] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. 2017. Compositional blocks for optimal self-healing gradients. In *Proceedings of the 11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 91–100. DOI: <https://doi.org/10.1109/SASO.2017.18>
- [3] Giorgio Audrito, Ferruccio Damiani, and Mirko Viroli. 2017. Optimally-self-healing distributed gradient structures through bounded information speed. In *Proceedings of the Coordination Models and Languages, Proceedings (COORDINATION'17)*, Lecture Notes in Computer Science, Vol. 10319. Springer, 59–77. DOI: https://doi.org/10.1007/978-3-319-59746-1_4
- [4] Giorgio Audrito, Ferruccio Damiani, and Mirko Viroli. 2018. Optimal single-path information propagation in gradient-based algorithms. *Sci. Comput. Program.* 166 (2018), 146–166. DOI: <https://doi.org/10.1016/j.scico.2018.06.002>
- [5] Jos C. M. Baeten, Twan Basten, Twan Basten, and M. A. Reniers. 2010. *Process Algebra: Equational Theories of Communicating Processes*. Vol. 50. Cambridge University Press. DOI: <https://doi.org/10.1017/CBO9781139195003>
- [6] Jacob Beal. 2009. Dynamically defined processes for spatial computers. In *Proceedings of the Spatial Computing Workshop*. IEEE, New York, NY, 206–211. DOI: <https://doi.org/10.1109/SASOW.2010.74>
- [7] Jacob Beal and Jonathan Bachrach. 2006. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intell. Syst.* 21, 2 (Mar./Apr. 2006), 10–19. DOI: <https://doi.org/10.1109/MIS.2006.29>
- [8] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. 2013. Organizing the aggregate: Languages for spatial computing. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, Marjan Mernik (Ed.). IGI Global, 436–501. DOI: <https://doi.org/10.4018/978-1-4666-2092-6.ch016>
- [9] Jacob Beal, Danilo Pianini, and Mirko Viroli. 2015. Aggregate programming for the internet of things. *IEEE Comput.* 48, 9 (2015). DOI: <https://doi.org/10.1109/MC.2015.261>
- [10] Jacob Beal and Mirko Viroli. 2014. Building blocks for aggregate programming of self-organising applications. In *Proceedings of the 8th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. IEEE, 8–13. DOI: <https://doi.org/10.1109/SASOW.2014.6>
- [11] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. 2016. Self-adaptation to device distribution changes in situated computing systems. In *Proceedings of the IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO'16)*. IEEE Computer Society, 60–69. DOI: <https://doi.org/10.1109/SASO.2016.12>
- [12] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. 2017. Self-adaptation to device distribution in the internet of things. *ACM Trans. Auton. Adapt. Syst.* 12, 3 (Sep. 2017), 12:1–12:29. DOI: <https://doi.org/10.1145/3105758>
- [13] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gian Luigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. 2003. The Klaim project: Theory and practice. In *Proceedings of*

- the Global Computing 2003*, Lecture Notes in Computer Science, Vol. 2874. Springer, 88–150. DOI : https://doi.org/10.1007/978-3-540-40042-4_4
- [14] William Butera. 2002. *Programming a Paintable Computer*. Ph.D. Dissertation. MIT, Cambridge.
 - [15] Luca Cardelli and Philippa Gardner. 2010. Processes in space. In *Proceedings of the 6th Conference on Computability in Europe*, Lecture Notes in Computer Science, Vol. 6158. Springer, 78–87. DOI : <https://doi.org/10.1007/978-3-642-13962-8>
 - [16] Luca Cardelli and Andrew D. Gordon. 2000. Mobile ambients. *Theor. Comput. Sci.* 240, 1 (Jun. 2000), 177–213. DOI : [https://doi.org/10.1016/S0304-3975\(99\)00231-5](https://doi.org/10.1016/S0304-3975(99)00231-5)
 - [17] Alonzo Church. 1932. A set of postulates for the foundation of logic. *Ann. Math.* 33, 2 (1932), 346–366. DOI : <https://doi.org/10.2307/1968337>
 - [18] Lauren Clement and Radhika Nagpal. 2003. Self-assembly and self-repairing topologies. In *Proceedings of the Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*.
 - [19] Daniel Coore. 1999. *Botanical Computing: A Developmental Approach to Generating Inter Connect Topologies on an Amorphous Computer*. Ph.D. Dissertation. MIT, Cambridge, MA.
 - [20] Pierre-Louis Curien. 2007. Definability and full abstraction. *Electr. Notes Theor. Comput. Sci.* 172 (2007), 301–310. DOI : <https://doi.org/10.1016/j.entcs.2007.02.011>
 - [21] Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'82)*. ACM, 207–212. DOI : <https://doi.org/10.1145/582153.582176>
 - [22] Ferruccio Damiani and Mirko Viroli. 2015. Type-based self-stabilisation for computational fields. *Logical Methods Comput. Sci.* 11, 4 (Dec. 2015). DOI : [https://doi.org/10.2168/LMCS-11\(4:21\)2015](https://doi.org/10.2168/LMCS-11(4:21)2015)
 - [23] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. 2016. A type-sound calculus of computational fields. *Sci. Comput. Program.* 117 (2016), 17–44. DOI : <https://doi.org/10.1016/j.scico.2015.11.005>
 - [24] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. 2015. Code mobility meets self-organisation: A higher-order calculus of computational fields. In *Proceedings of the Formal Techniques for Distributed Objects, Components, and Systems (FORTE'15)*, Lecture Notes in Computer Science, Vol. 9039. Springer, 113–128. DOI : https://doi.org/10.1007/978-3-319-19195-9_8
 - [25] J. W. de Bakker and Jeffery I. Zucker. 1982. Denotational semantics of concurrency. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*. ACM, 153–158. DOI : <https://doi.org/10.1145/800070.802188>
 - [26] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. 2013. A language-based approach to autonomic computing. In *Proceedings of the Formal Methods for Components and Objects (FMCO'11)*, Lecture Notes in Computer Science, Vol. 7542. Springer, 25–48. DOI : https://doi.org/10.1007/978-3-642-35887-6_2
 - [27] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113. DOI : <https://doi.org/10.1145/1327452.1327492>
 - [28] Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluís Arcos. 2013. Description and composition of bio-inspired design patterns: A complete overview. *Nat. Comput.* 12, 1 (2013), 43–67. DOI : <https://doi.org/10.1007/s11047-012-9324-y>
 - [29] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. 1994. KQML as an agent communication language. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*. ACM, 456–463. DOI : <https://doi.org/10.1145/191246.191322>
 - [30] William I. Gasarch. 2015. Proving programs terminate using well-founded orderings, Ramsey's theorem, and matrices. *Adv. Comput.* 97 (2015), 147–200. DOI : <https://doi.org/10.1016/bs.adcom.2014.12.002>
 - [31] David Gelernter. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.* 7, 1 (1985), 80–112. DOI : <https://doi.org/10.1145/2363.2433>
 - [32] Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Przemyslaw Prusinkiewicz. 2002. *Computational Models for Integrative and Developmental Biology*. Technical Report 72-2002. Univerite d'Evry, LaMI.
 - [33] J. Roger Hindley. 1997. *Basic Simple Type Theory*. Cambridge University Press. DOI : <https://doi.org/10.1017/CBO9780511608865>
 - [34] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. DOI : <https://doi.org/10.1145/503502.503505>
 - [35] M. E. Inchiosa and M. T. Parker. 2002. Overcoming design and development challenges in agent-based modeling using ASCAPE. *Proc. Natl. Acad. Sci. U.S.A.* 99, 3 (2002), 7304. DOI : <https://doi.org/10.1073/pnas.082081199>
 - [36] Shay Kutten and Boaz Patt-Shamir. 1997. Time-adaptive self stabilization. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 149–158. DOI : <https://doi.org/10.1145/259380.259435>
 - [37] Frank C. H. Lin and Robert M. Keller. 1987. The gradient model load balancing method. *IEEE Trans. Softw. Eng.* 13, 1 (1987), 32–38. DOI : <https://doi.org/10.1109/TSE.1987.232563>
 - [38] Samuel R. Madden, Robert Szewczyk, Michael J. Franklin, and David Culler. 2002. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Proceedings of the Workshop on Mobile Computing and Systems Applications*. IEEE. DOI : <https://doi.org/10.1109/MCSA.2002.1017485>

- [39] Marco Mamei and Franco Zambonelli. 2009. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. Softw. Eng. Method.* 18, 4 (2009), 1–56. DOI : <https://doi.org/10.1145/1538942.1538945>
- [40] José Meseguer, Joseph A. Goguen, and Gert Smolka. 1989. Order-sorted unification. *J. Symbol. Comput.* 8, 4 (1989), 383–413. DOI : [https://doi.org/10.1016/S0747-7171\(89\)80036-7](https://doi.org/10.1016/S0747-7171(89)80036-7)
- [41] Message Passing Interface Forum 2009. *MPI: A Message-Passing Interface Standard Version 2.2*. Message Passing Interface Forum.
- [42] Robin Milner. 1997. *The Definition of Standard ML: Revised*. MIT press.
- [43] Robin Milner. 2006. Pure bigraphs: Structure and dynamics. *Inf. Comput.* 204, 1 (2006), 60–122. DOI : <https://doi.org/10.1016/j.ic.2005.07.003>
- [44] Radhika Nagpal. 2001. *Programmable Self-Assembly: Constructing Global Shape Using Biologically-inspired Local Interactions and Origami Mathematics*. Ph.D. Dissertation. MIT, Cambridge, MA.
- [45] Ryan Newton and Matt Welsh. 2004. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the Workshop on Data Management for Sensor Networks*. ACM, 78–87. DOI : <https://doi.org/10.1145/1052199.1052213>
- [46] Yuichi Nishiwaki. 2016. Digamma-calculus: A universal programming language of self-stabilizing computational fields. In *Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (eCAS'16)*. IEEE. DOI : <https://doi.org/10.1109/FAS-W.2016.51>
- [47] Atsushi Ohori. 1989. A simple semantics for ML polymorphism. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM, 281–292. DOI : <https://doi.org/10.1145/99370.99393>
- [48] C.-H. Luke Ong. 1995. Correspondence between operational and denotational semantics: The full abstraction problem for PCF. *Handbook of Logic in Computer Science* 4 (1995), 269–356.
- [49] Danilo Pianini, Sara Montagna, and Mirko Viroli. 2013. Chemical-oriented simulation of computational systems with Alchemist. *J. Simul.* 7, 3 (2013), 202–215. DOI : <https://doi.org/10.1057/jos.2012.27>
- [50] Danilo Pianini, Mirko Viroli, and Jacob Beal. 2015. Protelis: Practical aggregate programming. In *Proceedings of the ACM Symposium on Applied Computing 2015*. ACM, 1846–1853. DOI : <https://doi.org/10.1145/2695664.2695913>
- [51] Elizabeth Sklar. 2007. NetLogo, a multi-agent simulation environment. *Artif. Life* 13, 3 (2007), 303–311. DOI : <https://doi.org/10.1162/artl.2007.13.3.303>
- [52] Allen Stoughton. 1988. *Fully Abstract Models of Programming Languages*. Pitman.
- [53] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. 2018. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.* 28, 2 (2018), 16:1–16:28. DOI : <https://doi.org/10.1145/3177774>
- [54] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. 2018. From field-based coordination to aggregate computing. In *Proceedings of the Coordination Models and Languages (COORDINATION'18)*, Lecture Notes in Computer Science, Vol. 10852. Springer, 252–279. DOI : https://doi.org/10.1007/978-3-319-92408-3_12
- [55] Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. 2011. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Trans. Auton. Adapt. Syst.* 6, 2 (Jun. 2011), 14:1–14:24. DOI : <https://doi.org/10.1145/1968513.1968517>
- [56] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. 2013. A calculus of computational fields. In *Proceedings of the Advances in Service-Oriented and Cloud Computing (Communications in Computer and Information Science)*, Vol. 393. Springer, 114–128. DOI : https://doi.org/10.1007/978-3-642-45364-9_11
- [57] Mirko Viroli, Danilo Pianini, and Jacob Beal. 2012. Linda in space-time: An adaptive coordination model for mobile ad-hoc environments. In *Proceedings of the Coordination Models and Languages (COORDINATION'12)*, Lecture Notes in Computer Science, Vol. 7274. Springer, 212–229. DOI : https://doi.org/10.1007/978-3-642-30829-1_15
- [58] Mirko Viroli, Danilo Pianini, Sara Montagna, Graeme Stevenson, and Franco Zambonelli. 2015. A coordination model of pervasive service ecosystems. *Sci. Comput. Program.* 110 (2015), 3–22. DOI : <https://doi.org/10.1016/j.scico.2015.06.003>
- [59] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. 2004. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*. ACM Press. DOI : <https://doi.org/10.1145/990064.990079>
- [60] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages—An Introduction*. MIT Press.

Received October 2016; revised October 2018; accepted October 2018