Logical Methods in Computer Science
Vol. 11(4:21)2015, pp. 1–53
www.lmcs-online.org

Submitted Dec. 15, 2014
Published Dec. 31, 2015

TYPE-BASED SELF-STABILISATION FOR COMPUTATIONAL FIELDS *

FERRUCCIO DAMIANI [a] AND MIRKO VIROLI [b]

[a] University of Torino, Italy
e-mail address: ferruccio.damiani@unito.it

[b] University of Bologna, Italy
e-mail address: mirko.viroli@unibo.it

ABSTRACT. Emerging network scenarios require the development of solid large-scale situated systems. Unfortunately, the diffusion/aggregation computational processes therein often introduce a source of complexity that hampers predictability of the overall system behaviour. Computational fields have been introduced to help engineering such systems: they are spatially distributed data structures designed to adapt their shape to the topology of the underlying (mobile) network and to the events occurring in it, with notable applications to pervasive computing, sensor networks, and mobile robots. To assure behavioural correctness, namely, correspondence of micro-level specification (single device behaviour) with macro-level behaviour (resulting global spatial pattern), we investigate the issue of self-stabilisation for computational fields. We present a tiny, expressive, and type-sound calculus of computational fields, and define sufficient conditions for self-stabilisation, defined as the ability to react to changes in the environment finding a new stable state in finite time. A type-based approach is used to provide a correct checking procedure for self-stabilisation.

1. INTRODUCTION

Computational fields [34, 42] (sometimes simply *fields* in the following) are an abstraction traditionally used to enact self-organisation mechanisms in contexts including swarm robotics [3], sensor

2012 ACM CCS: [**Theory of computation**]: Models of computation—Concurrency—Distributed computing models; Logic—Type theory; Design and analysis of algorithms—Distributed algorithms—Self-organization; Semantics and reasoning—Program constructs—Control primitives; Semantics and reasoning—Program semantics; Semantics and reasoning—Program reasoning—Program analysis; [**Software and its engineering**]: Software notations and tools — General programming languages—Language types— Distributed programming languages / Functional languages; Software notations and tools—Context specific languages; [**Computing methodologies**]: Artificial intelligence—Distributed artificial intelligence—Cooperation and coordination.

Key words and phrases: Computational field, Core calculus, Operational semantics, Spatial computing, Type-based analysis, Type soundness, Type system, Refinement type.

* A preliminary version of some of the material presented in this paper has appeared in COORDINATION 2015.

[a] Ferruccio Damiani has been partially supported by project HyVar (*www.hyvar-project.eu*) which 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*), by ICT COST Action IC1201 BETTY (*www.behavioural-types.eu*), by the Italian PRIN 2010/2011 project CINA (*sysma.imtlucca.it/cina*), and by Ateneo/CSP project SALT (*salt.di.unito.it*).

[b] Mirko Viroli has been partially supported by the EU FP7 project "SAPERE - Self-aware Pervasive Service Ecosystems" under contract No. 256873 and by the Italian PRIN 2010/2011 project CINA (*sysma.imtlucca.it/cina*).

LOGICAL METHODS IN COMPUTER SCIENCE

DOI:10.2168/LMCS-11(4:21)2015

© F. Damiani and M. Viroli
Creative Commons

networks [5], pervasive computing [35], task assignment [49], and traffic control [13]. They are distributed data structures originated from pointwise events raised in some specific device (i.e., a sensor), and propagating in a whole network region until forming a spatio-temporal data structure upon which distributed and coordinated computation can take place. Middleware/platforms supporting this notion include TOTA [35], Proto [36], and SAPERE [51, 44]. The most paradigmatic example of computational field is the so-called *gradient* [6, 35, 38], mapping each node of the network to the minimum distance from the source node where the gradient has been injected. Gradients are key to get awareness of physical/logical distances, to project a single-device event into a whole network region, and to find the direction towards certain locations of a network, e.g., for routing purposes. Several pieces of work have been developed that investigate coordination models supporting fields [35, 47], introduce advanced gradient-based spatial patterns [37], study universality and expressiveness [10], and develop catalogues of self-organisation mechanisms where gradients play a crucial role [25].

As with most self-organisation approaches, a key issue is to try to fill the gap between the system micro-level (the single-node computation and interaction behaviour) and the system macro-level (the shape of the globally established spatio-temporal structure), namely, ensuring that the programmed code results in the expected global-level behaviour. However, the issue of formally tackling the problem is basically yet unexplored in the context of spatial computing, coordination, and process calculi—some exceptions are [6, 27], which however apply in rather ad-hoc cases. We note instead that studying this issue will likely shed light on which language constructs are best suited for developing well-engineered self-organisation mechanisms based on computational fields, and to consolidate existing patterns or develop new ones.

In this paper we follow this direction and devise an expressive calculus to specify the propagation process of those computational fields for which we can identify a precise mapping between system micro- and macro-level. The key constructs of the calculus are three: sensor fields (considered as an environmental input), pointwise functional composition of fields, and a form of *spreading* that tightly couples information diffusion and re-aggregation. The spreading construct is constrained so as to enforce a special "stabilising-diffusion condition" that we identified, by which we derive self-stabilisation [22], that is, the ability of the system running computational fields to reach a stable distributed state in spite of perturbations (changes of network topology and of local sensed data) from which it recovers in finite time. A consequence of our results is that the ultimate (and stable) state of an even complex computational field can be fully-predicted once the environment state is known (network topology and sensors state). Still, checking that a field specification satisfies such stabilising-diffusion condition is subtle, since it involves the ability of reasoning about the relation holding between inputs and outputs of functions used to propagate information across nodes. Hence, as an additional contribution, we introduce a type-based approach that provides a correct checking procedure for the stabilising-diffusion condition.

The remainder of this paper is organised as follows: Section 2 illustrates the proposed linguistic constructs by means of examples; Section 3 presents the calculus and formalises the self-stabilisation property; Section 4 introduces the stabilising-diffusion condition to constrain spreading in order to ensure self-stabilisation; Section 5 proves that the stabilising-diffusion condition guarantees self-stabilisation; Section 6 extends the calculus with the pair data structure and provides further examples; Sections 7, 8 and 9 incrementally present a type-based approach for checking the stabilising-diffusion condition and prove that the approach is sound; Section 10 discusses related work; and finally Section 11 concludes and discusses directions for future work. The appendices contain the proof of the main results. A preliminary version of some of the material presented in this paper appeared in [45].

$\mathtt{e}$	$::=$	$\mathtt{x} \mid s \mid \mathtt{g} \mid \mathtt{e}_0 ? \mathtt{e}_1 : \mathtt{e}_2 \mid \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \mid \{\mathtt{e} : \mathtt{f}(@, \mathtt{e}_1, \ldots, \mathtt{e}_n)\}$	expression
$\mathtt{f}$	$::=$	$\mathtt{d} \mid \mathtt{b}$	function name
$\mathtt{D}$	$::=$	$\mathtt{def}\ \mathtt{T}\ \mathtt{d}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{is}\ \mathtt{e}$	function definition

Figure 1: Syntax of expressions and function definitions

2. Computational Fields

From an abstract viewpoint, a computational field is simply a map from nodes of a network to some kind of value. They are used as a valuable abstraction to engineer self-organisation into networks of situated devices. Namely, out of local interactions (devices communicating with a small neighbourhood), global and coherent patterns (the computational fields themselves) establish that are robust to changes of environmental conditions. Such an adaptive behaviour is key in developing system coordination in dynamic and unpredictable environments [40].

Self-organisation and computational fields are known to build on top of three basic mechanisms [25]: diffusion (devices broadcast information to their neighbours), aggregation (multiple information can be reduced back into a single sum-up value), and evaporation/decay (a cleanup mechanism is used to reactively adapt to changes). These mechansisms have been used to synthetise a rather vast set of distributed algorithms [21, 52, 25, 7].

For instance, these mechanisms are precisely those used to create adaptive and stable *gradients*, which are building blocks of more advanced patterns [25, 37]. A gradient is used to reify in any node some information about the path towards the nearest gradient source. It can be computed by the following process: value 0 is held in the gradient source; each node executes asynchronous computation rounds in which *(i)* messages from neighbours are gathered and *aggregated* in a minimum value, *(ii)* this is increased by one and is *diffused* to all neighbours, and *(iii)* the same value is stored locally, to replace the old one which *decays*. This continuous "spreading process" stabilises to a so called *hop-count gradient*, storing distance to the nearest source in any node, and automatically repairing in finite time to changes in the environment (changes of topology, position and number of sources).

2.1. Basic Ingredients.

Based on these ideas, and framing them so as to isolate those cases where the spreading process actually stabilises, we propose a core calculus to express computational fields. Its syntax is reported in Figure 1. Our language is typed and (following the general approach used in other languages for spatial computing [46, 36], which the one we propose here can be considered as a core fragment) functional.

Types $\mathtt{T}$ are monomorphic. For simplicity, only *ground types* $\mathtt{G}$ (like `real` and `bool`) are modeled—in Section 6 we will point out that the properties of the calculus are indeed parametric in the set of modeled types (in particular, we will consider and extension of the calculus that models pairs). We write $[\![\mathtt{T}]\!]$ to denote the set of the values of type $\mathtt{T}$. We assume that each type $\mathtt{T}$ is equipped with a *total order* $\leq_{\mathtt{T}}$ over $[\![\mathtt{T}]\!]$ that is *noetherian* [31], i.e., there are no infinite ascending chains of values $\mathtt{v}_0 <_{\mathtt{T}} \mathtt{v}_1 <_{\mathtt{T}} \mathtt{v}_2 <_{\mathtt{T}} \cdots$. This implies that $[\![\mathtt{T}]\!]$ has a maximum element, that we denote by $\top_{\mathtt{T}}$. Each ground type usually comes with a natural ordering (for `bool` we consider $\mathtt{FALSE} \leq_{\mathtt{bool}} \mathtt{TRUE}$) which is total and noetherian—though in principle ad-hoc ordering relations could be used in a deployed specification language.

An expression can be a *variable* $\mathtt{x}$, a *sensor* s, a *ground-value* $\mathtt{g}$, a *conditional* $\mathtt{e}_0 ? \mathtt{e}_1 : \mathtt{e}_2$, a *function application* $\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ or a *spreading* $\{\mathtt{e} : \mathtt{f}(@, \mathtt{e}_1, \ldots, \mathtt{e}_n)\}$. Variables are the formal parameters of a function.

Sensors are sources of input produced by the environment, available in each device (in examples, we shall use for them literals starting with symbol "#"). For instance, in a urban scenario we may want to use a crowd sensor `#crowd` yielding non-negative real numbers, to represent the perception of crowd level available in each deployed sensor over time [38].

Values `v` coincide with *ground values* `g` (i.e., values of ground type), like reals (e.g., `1`, `5.7`,... and `POSINF` and `NEGINF` meaning the maximum and the minimum real, respectively) and booleans (`TRUE` and `FALSE`).

A function can be either a built-in function `b` or a user-defined function `d`. Built-in functions include usual mathematical/logical ones, used either in prefix or infix notation, e.g., to form expressions like `2*#crowd` and `or(TRUE,FALSE)`. User-defined functions are declared by a *function definition* $\mathtt{def}\ \mathtt{T}\ \mathtt{d}(\mathtt{T}_1\ \mathtt{x}_1,...,\mathtt{T}_n\ \mathtt{x}_n)$ `is e`—cyclic definitions are prohibited, and the 0-ary function `main` is the program entry point. As a first example of user-defined function consider the following function `restrict`:

```
def real restrict(real i, bool c) is c ?  i :  POSINF.
```

It takes two arguments `i` and `c`, and yields the former if `c` is true, or `POSINF` otherwise—as we shall see, because of our semantics `POSINF` plays a role similar to an undefined value.

A *pure function* `f` is either a built-in function `b`, or a user-defined function `d` whose call graph (including `d` itself) does not contain functions with spreading expressions or sensors in their body. We write $[\![\mathtt{f}]\!]$ to denote the (trivial) semantics of a pure-function `f`, which is a computable functions that maps a tuple of elements from $[\![\mathtt{T}_1]\!]$,..., $[\![\mathtt{T}_n]\!]$ to $[\![\mathtt{T}]\!]$, where $\mathtt{T}_1,...,\mathtt{T}_n$ and `T` are the types of the arguments and the type of the result of `f`, respectively.

As in [46, 36], expressions in our language have a twofold interpretation. When focussing on the *local* device behaviour, they represent values computed in a node at a given time. When reasoning about the *global* outcome of a specification instead, they represent whole computational fields: 1 is the immutable field holding 1 in each device, `#crowd` is the (evolving) crowd field, and so on.

The key construct of the proposed language is *spreading*, denoted by syntax $\{\mathtt{e} : \mathtt{f}(@,\mathtt{e}_1,...,\mathtt{e}_n)\}$, where `e` is called *source* expression, and $\mathtt{f}(@,\mathtt{e}_1,...,\mathtt{e}_n)$ is called *diffusion* expression. In a diffusion expression the function `f`, which we call *diffusion*, must be a pure function whose return type and first argument type are the same. The symbol `@` plays the role of a formal argument, hence the diffusion expression can be seen as the body of an anonymous, unary function. Viewed locally to a node, expression $\mathtt{e} = \{\mathtt{e}_0 : \mathtt{f}(@,\mathtt{e}_1,...,\mathtt{e}_n)\}$ is evaluated at a given time to value `v` as follows:

(1) expressions $\mathtt{e}_0,\mathtt{e}_1,...,\mathtt{e}_n$ are evaluated to values $\mathtt{v}_0,\mathtt{v}_1,...,\mathtt{v}_n$;
(2) the current values $\mathtt{w}_1,...,\mathtt{w}_m$ of `e` in neighbours are gathered;
(3) for each $\mathtt{w}_j$ in them, the diffusion function is applied as $\mathtt{f}(\mathtt{w}_j,\mathtt{v}_1,...,\mathtt{v}_n)$, giving value $\mathtt{w}'_j$;
(4) the final result `v` is the minimum value among $\{\mathtt{v}_0,\mathtt{w}'_1,...,\mathtt{w}'_m\}$: this value is made available to other nodes.

Note that $\mathtt{v} \leq_{\mathtt{T}} \mathtt{v}_0$, and if the device is isolated then $\mathtt{v} = \mathtt{v}_0$. Viewed globally, $\{\mathtt{e}_0 : \mathtt{f}(@,\mathtt{e}_1,...,\mathtt{e}_n)\}$ represents a field initially equal to $\mathtt{e}_0$; as time passes some field values can decrease due to smaller values being received from neighbours (after applying the diffusion function).

The hop-count gradient created out of a `#src` sensor is hence simply defined as

```
{ #src :  @ + 1 }
```

assuming `#src` holds what we call a zero-field, namely, it is 0 on source nodes and `POSINF` everywhere else. In this case `#src` is the source expression, and `f` is unary successor function.

```
def real grad(real i) is { i :  @ + #dist }
def real restrict(real i, bool c) is c ?  i :  POSINF
def real restrictSum(real x, real y, bool c) is restrict(x + y, c)
def real gradobs(real i, bool c) is { i :  restrictSum(@,#dist, c) }
def float gradbound(real i, real z) is gradobs(i, grad(i) < z))
```

Figure 2: Definitions of examples

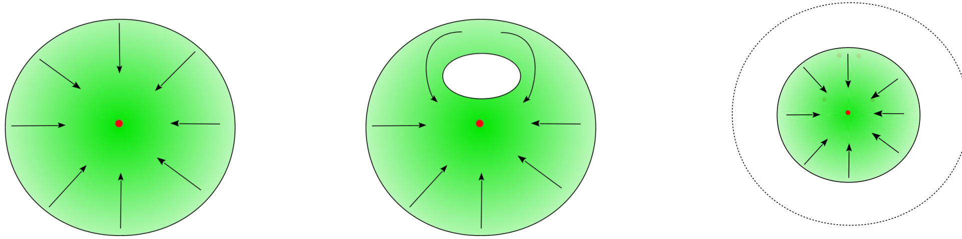

Figure 3: Pictorial representation of hop-count gradient field (left), a gradient circumventing "crowd" obstacles field (center), and a gradient with bounded distance (right)

2.2. **Examples.** As a reference scenario to ground the discussion, we can consider crowd steering in pervasive environments [38]: computational fields run on top of a myriad of small devices spread in the environment (including smartphones), and are used to guide people in complex environments (buildings, cities) towards points of interest (POIs) across appropriate paths. There, a smartphone can perceive neighbour values of a gradient spread from a POI, and give directions towards smallest values so as to steer its owner and make him/her quickly descend the gradient [35]. Starting from the hop-count gradient, various kinds of behaviour useful in crowd steering can be programmed, based on the definitions reported in Figure 2.

The first function in Figure 2 defines a more powerful gradient construct, called `grad`, which can be used to generalise over the hop-by-hop notion of distance: sensor `#dist` is assumed to exist that reifies an application-specific notion of distance as a positive number. It can be 1 everywhere to model hop-count gradient, or can vary from device to device to take into consideration contextual information. For instance, it can be the output of a crowd sensor, leading to greater distances when/where crowded areas are perceived, so as to dynamically compute routes penalising crowded areas as in [38]. In this case, note that diffusion function `f` maps $(\mathtt{v}_1, \mathtt{v}_2)$ to $\mathtt{v}_1 + \mathtt{v}_2$. Figure 3 (left) shows a pictorial representation, assuming devices are uniformly spread in a 2D environment: considering that an agent or data items move in the direction descending the values of a field, a gradient looks like a sort of uniform attractor towards the source, i.e., to the nearest source node. It should be noted that when deployed in articulated environments, the gradient would stretch and dilate to accommodate the static/dynamic shape of environment, computing optimal routes.

By suitably changing the diffusion function, it is also possible to block the diffusion process of gradients, as shown in function `gradobs`: there, by restriction we turn the gradient value to `POSINF` in nodes where the "obstacle" boolean field `c` holds `FALSE`. This can be used to completely circumvent obstacle areas, as shown in Figure 3 (center). Note that we here refer to a "blocking" behaviour, since sending a `POSINF` value has no effect on the target because of the semantics of spreading; hence, an optimised implementation could simply avoid sending a `POSINF` at all, so as

Expression type checking: $\boxed{\mathscr{T} \vdash \mathtt{e} : \mathtt{T}}$

$$\text{[T-VAR]}\ \frac{}{\mathscr{T}, \mathtt{x} : \mathtt{T} \vdash \mathtt{x} : \mathtt{T}} \qquad \text{[T-SNS]}\ \frac{}{\mathscr{T} \vdash s : \mathit{type}(s)} \qquad \text{[T-GVAL]}\ \frac{}{\mathscr{T} \vdash \mathtt{g} : \mathit{type}(\mathtt{g})}$$

$$\text{[T-COND]}\ \frac{\mathscr{T} \vdash \mathtt{e}_0 : \mathtt{bool} \quad \mathscr{T} \vdash \mathtt{e}_1 : \mathtt{T} \quad \mathscr{T} \vdash \mathtt{e}_2 : \mathtt{T}}{\mathscr{T} \vdash \mathtt{e}_0 ? \mathtt{e}_1 : \mathtt{e}_2 : \mathtt{T}}$$

$$\text{[T-FUN]}\ \frac{\mathtt{T}(\overline{\mathtt{T}}) = \mathit{t\text{-}sig}(\mathtt{f}) \quad \mathscr{T} \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}}{\mathscr{T} \vdash \mathtt{f}(\overline{\mathtt{e}}) : \mathtt{T}} \qquad \text{[T-SPR]}\ \frac{\mathit{diffusion}(\mathtt{f}) \quad \mathscr{T} \vdash \mathtt{f}(\mathtt{e}, \overline{\mathtt{e}}) : \mathtt{T}}{\mathscr{T} \vdash \{\mathtt{e} : \mathtt{f}(\mathtt{@}, \overline{\mathtt{e}})\} : \mathtt{T}}$$

User-defined function type checking: $\boxed{\vdash \mathtt{D} : \mathtt{T}(\overline{\mathtt{T}})}$

$$\text{[T-DEF]}\ \frac{\overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}}{\vdash \mathtt{def}\ \mathtt{T}\ \mathtt{d}(\overline{\mathtt{T}}\ \overline{\mathtt{x}}) = \mathtt{e} : \mathtt{T}(\overline{\mathtt{T}})}$$

Figure 4: Type-checking rules for expressions and function definitions

not to flood the entire network. This pattern is useful whenever steering people in environments with prohibited areas—e.g. road construction in a urban scenario.

Finally, by a different blocking mechanism we can limit the propagation distance of a gradient, as shown by function `gradbound` and Figure 3 (right): the second argument `z` imposes a numerical bound to the distance, which is applied by exploiting the functions `gradobs` and `grad`.

In section 6 we will exploit the pair data structure to program more advanced examples of behaviour useful in crowd steering.

3. The Calculus of Self-Stabilising Computational Fields

After informally introducing the proposed calculus in previous section, we now provide a formal account of it and precisely state the self-stabilisation property. Namely, we formalise and illustrate by means of examples the type system (in Section 3.1), the operational semantics (in Section 3.2), and the self-stabilisation property (in Section 3.3).

3.1. **Type checking.** The syntax of the calculus is reported in Figure 1. As a standard syntactic notation in calculi for object-oriented and functional languages [32], we use the overbar notation to denote metavariables over lists, e.g., we let $\overline{\mathtt{e}}$ range over lists of expressions, written $\mathtt{e}_1\ \mathtt{e}_2\ ...\ \mathtt{e}_n$, and similarly for $\overline{\mathtt{x}}$, $\overline{\mathtt{T}}$ and so on. We write $\mathit{t\text{-}sig}(\mathtt{f})$ to denote the type-signature $\mathtt{T}(\overline{\mathtt{T}})$ of `f` (which specifies the type `T` of the result and the types $\overline{\mathtt{T}} = \mathtt{T}_1, ..., \mathtt{T}_n$ of the $n \geq 0$ arguments of `f`). We assume that the mapping $\mathit{t\text{-}sig}(\cdot)$ associates a type-signature to each built-in function and, for user-defined functions, returns the type-signature specified in the function definition.

A program **P** in our language is a mapping from function names to function definitions, enjoying the following *sanity conditions*: *(i)* $\mathbf{P}(\mathtt{d}) = \mathtt{def}\ \mathtt{d} \cdots (\cdots)\ \mathtt{is} \cdots$ for every $\mathtt{d} \in \mathit{dom}(\mathbf{P})$; *(ii)* for every function name `d` appearing anywhere in **P**, we have $\mathtt{d} \in \mathit{dom}(\mathbf{P})$; *(iii)* there are no cycles in the function call graph (i.e., there are no recursive functions in the program); and *(iv)* $\mathtt{main} \in \mathit{dom}(\mathbf{P})$ and it has zero arguments. A program that does not contain the `main` function is called a *library*.

The type system we provide aims to guarantee that no run-time error may arise during evaluation: its typing rules are given in Figure 4. *Type environments*, ranged over by $\mathscr{T}$ and written $\overline{\mathtt{x}} : \overline{\mathtt{T}}$, contain type assumptions for program variables. The type-checking judgement for expressions is of the form

Ground types:

$\mathtt{G} \; = \; \mathtt{bool} \;\big|\; \mathtt{real}$

Sensor types:

$$\begin{array}{lcl} type(\mathtt{\#src}) & = & \mathtt{real} \\ type(\mathtt{\#dist}) & = & \mathtt{real} \end{array}$$

Built-in function type-signatures:

$$\begin{array}{lcl} t\text{-}sig(\mathtt{not}) & = & \mathtt{bool}(\mathtt{bool}) \\ t\text{-}sig(\mathtt{or}) & = & \mathtt{bool}(\mathtt{bool},\mathtt{bool}) \\ t\text{-}sig(-) & = & \mathtt{real}(\mathtt{real}) \\ t\text{-}sig(+) & = & \mathtt{real}(\mathtt{real},\mathtt{real}) \\ t\text{-}sig(=) & = & \mathtt{bool}(\mathtt{real},\mathtt{real}) \\ t\text{-}sig(<) & = & \mathtt{bool}(\mathtt{real},\mathtt{real}) \end{array}$$

Figure 5: Ground types, types for sensors, and type-signatures for built-in functions used in the examples

$\mathscr{T} \vdash \mathtt{e} : \mathtt{T}$, to be read: $\mathtt{e}$ has type $\mathtt{T}$ under the type assumptions $\mathscr{T}$ for the program variables occurring in $\mathtt{e}$. As a standard syntax in type systems [32], given $\overline{\mathtt{x}} = \mathtt{x}_1,...,\mathtt{x}_n$, $\overline{\mathtt{T}} = \mathtt{T}_1,...,\mathtt{T}_n$ and $\overline{\mathtt{e}} = \mathtt{e}_1,...,\mathtt{e}_n$ ($n \geq 0$), we write $\overline{\mathtt{x}} : \overline{\mathtt{T}}$ as short for $\mathtt{x}_1 : \mathtt{T}_1,...,\mathtt{x}_n : \mathtt{T}_n$, and $\mathscr{T} \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$ as short for $\mathscr{T} \vdash \mathtt{e}_1 : \mathtt{T}_1 \cdots \mathscr{T} \vdash \mathtt{e}_n : \mathtt{T}_n$.

Type checking of variables, sensors, ground values, conditionals, and function applications are almost standard. In particular, values and sensors and built-in functions are given a type by construction: the mapping $type(\cdot)$ associates a sort to each ground value and to each sensor, while rule [T-FUN] exploits the mapping $t\text{-}sig(\cdot)$.

Example 3.1. Figure 5 illustrates the ground types, sensors, and built-in functions used in the examples introduced throughout the paper.

The only ad-hoc type checking is provided for spreading expressions $\{\mathtt{e} : \mathtt{f}(\mathtt{@},\overline{\mathtt{e}})\}$: they are given the type of $\mathtt{f}(\mathtt{e},\overline{\mathtt{e}})$, though the function $\mathtt{f}$ must be a *diffusion*, according to the following definition.

Definition 3.2 (Diffusion)**.** A type signature $\mathtt{T}(\overline{\mathtt{T}})$ with $\overline{\mathtt{T}} = \mathtt{T}_1,...,\mathtt{T}_n$ ($n \geq 1$) is a *diffusion type signature* (notation *diffusion*$(\mathtt{T}(\overline{\mathtt{T}}))$) if $\mathtt{T} = \mathtt{T}_1$. A pure function $\mathtt{f}$ is a *diffusion* (notation *diffusion*$(\mathtt{f})$) if its type signature $t\text{-}sig(\mathtt{f})$ is a diffusion type signature.

Example 3.3. Consider the functions defined in Figure 2. The following two predicates hold:
- *diffusion*$(\mathtt{+})$, and
- *diffusion*$(\mathtt{restrictSum})$

Function type checking, represented by judgement $\mathscr{T} \vdash \mathtt{D} : \mathtt{T}(\overline{\mathtt{T}})$, is standard. In the following we always consider a *well-typed* program (or library) **P**, to mean that all the function declarations in **P** type check. Note that no choice may be done when building a derivation for a given type-checking judgment, so the type-checking rules straightforwardly describe a type-checking algorithm.

Example 3.4. The library in Figure 2 type checks by using the ground types, sensors, and type-signatures for built-in functions in Figure 5.

3.2. **Operational Semantics.** In this section we fomalise the operational semantics of the calculus. As for the field calculus [46] and the Proto language [36], devices undergo computation in rounds. In each round, a device sleeps for some time, wakes up, gathers information about messages received from neighbours while sleeping, evaluates the program, and finally broadcasts a message to neighbours with information about the outcome of evaluation and goes back to sleep. The scheduling of such rounds across the network is fair and non-synchronous. The structure of the network may change over time: when a device is sleeping its neighborhood may change and the device itself may disappear (switch-off) and subsequently appear (switch-on). We first focus on single-device computations (in Section 3.2.1) and then on whole network evolution (in Section 3.2.2).

3.2.1. *Device Computation.* In the following, we let meta-variable ι range over the denumerable set $\mathbf{I}$ of *device identifiers*, meta-variable I over finite sets of such devices, meta-variables $\mathtt{u}$, $\mathtt{v}$ and $\mathtt{w}$ over values. Given a finite nonempty set $V \subseteq [\![\mathtt{T}]\!]$ we denote by $\bigwedge V$ its minimum element, and write $\mathtt{v} \wedge \mathtt{v}'$ as short for $\bigwedge\{\mathtt{v},\mathtt{v}'\}$.

In order to simplify the notation, we shall assume a fixed program $\mathbf{P}$ and write $\mathtt{e}_{\mathtt{main}}$ to denote the body of the `main` function. We say that "device ι *fires*", to mean that expression $\mathtt{e}_{\mathtt{main}}$ is evaluated on device ι. The result of the evaluation is a *value-tree*, which is an ordered tree of values, tracking the value of any evaluated subexpression. Intuitively, such an evaluation is performed against the most recently received value-trees of current neighbours and the current value of sensors, and produces as result a new value-tree that is broadcasted to current neighbours for their firing. Note that considering simply a value (instead of a value-tree) as the outcome of the evaluation $\mathtt{e}_{\mathtt{main}}$ on a device ι would not be enough, since the evaluation of each spreading expression $\mathtt{e}$ occurring in $\mathtt{e}_{\mathtt{main}}$ requires the values (at the root of their sub-value-trees) produced by the most recent evaluation of $\mathtt{e}$ on neighbours of ι (c.f. Sect. 2).[1]

The syntax of value-trees is given in Figure 6, together with the definition of the auxiliary functions $\rho(\cdot)$ and $\pi_i(\cdot)$ for extracting the root value and the i-th subtree of a value-tree, respectively— also the extension of these functions to sequences of value-environments $\overline{\theta}$ is defined. We sometimes abuse the notation writing a value-tree with just the root as $\mathtt{v}$ instead of $\mathtt{v}()$. The state of sensors σ is a map from sensor names to values, modelling the inputs received from the external world. This is written $\overline{s} \triangleright \overline{\mathtt{v}}$ as an abuse of notation to mean $s_1 \triangleright \mathtt{v}_1,...,s_n \triangleright \mathtt{v}_n$. We shall assume that it is complete (it has a mapping for any sensor used in the program), and correct (each sensor s has a type written $type(s)$, and is mapped to a value of that type). For this map, and for the others to come, we shall use the following notations: $\sigma(s)$ is used to extract the value that s is mapped to, $\sigma[\sigma']$ is the map obtained by updating σ with all the associations $s \triangleright \mathtt{v}$ of σ' which do not escape the domain of σ (namely, only those such that σ is defined for s).

The computation that takes place on a single device is formalised by the big-step operational semantics rules given in Figure 6. The derived judgements are of the form $\sigma;\overline{\theta} \vdash \mathtt{e} \Downarrow \theta$, to be read "expression $\mathtt{e}$ evaluates to value-tree θ on sensor state σ and w.r.t. the value-trees $\overline{\theta}$", where:

- σ is the current sensor-value map, modelling the inputs received from the external world;
- $\overline{\theta}$ is the list of the value-trees produced by the most recent evaluation of $\mathtt{e}$ on the current device's neighbours;
- $\mathtt{e}$ is the closed expression to be evaluated;
- the value-tree θ represents the values computed for all the expressions encountered during the evaluation of $\mathtt{e}$— in particular $\rho(\theta)$ is the local value of field expression $\mathtt{e}$.

[1] Any implementation might massively compress the value-tree, storing only enough information for tracking the values of spreading expressions.

Value-trees and sensor-value maps:

$$\theta, \eta \quad ::= \quad \mathtt{v}(\overline{\theta}) \qquad \text{value-tree}$$

$$\sigma \quad ::= \quad \overline{s} \rhd \overline{\mathtt{v}} \qquad \text{sensor-value map}$$

Auxiliary functions:

$$\rho(\mathtt{v}(\overline{\theta})) = \mathtt{v} \qquad \pi_i(\mathtt{v}(\theta_1, ..., \theta_n)) = \theta_i$$

$$\rho(\theta_1, ..., \theta_n) = \rho(\theta_1), ..., \rho(\theta_n) \qquad \pi_i(\theta_1, ..., \theta_n) = \pi_i(\theta_1), ..., \pi_i(\theta_n)$$

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

$$\text{[E-SNS]} \frac{}{\sigma; \overline{\theta} \vdash s \Downarrow \sigma(s)} \qquad \text{[E-VAL]} \frac{}{\sigma; \overline{\theta} \vdash \mathtt{v} \Downarrow \mathtt{v}}$$

$$\text{[E-COND]} \frac{\sigma; \pi_1(\overline{\theta}) \vdash \mathtt{e}_1 \Downarrow \eta_1 \quad \cdots \quad \sigma; \pi_3(\overline{\theta}) \vdash \mathtt{e}_3 \Downarrow \eta_3 \qquad \mathtt{v} = \begin{cases} \rho(\eta_2) & \text{if } \rho(\eta_1) = \mathtt{TRUE} \\ \rho(\eta_3) & \text{if } \rho(\eta_1) = \mathtt{FALSE} \end{cases}}{\sigma; \overline{\theta} \vdash \mathtt{e}_1 ? \mathtt{e}_2 : \mathtt{e}_3 \Downarrow \mathtt{v}(\eta_1, \eta_2, \eta_3)}$$

$$\text{[E-BLT]} \frac{\sigma; \pi_1(\overline{\theta}) \vdash \mathtt{e}_1 \Downarrow \eta_1 \quad \cdots \quad \sigma; \pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \eta_n \qquad \mathtt{v} = [\![\mathtt{b}]\!](\rho(\eta_1), ..., \rho(\eta_n))}{\sigma; \overline{\theta} \vdash \mathtt{b}(\mathtt{e}_1, ..., \mathtt{e}_n) \Downarrow \mathtt{v}(\eta_1, ..., \eta_n)}$$

$$\text{[E-DEF]} \frac{\begin{array}{c} \mathtt{def}\ \mathtt{T}\ \mathtt{d}(\mathtt{T}_1\ \mathtt{x}_1, ..., \mathtt{T}_n\ \mathtt{x}_n) = \mathtt{e} \qquad \sigma; \pi_1(\overline{\theta}) \vdash \mathtt{e}_1 \Downarrow \theta'_1 \quad \cdots \quad \sigma; \pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \theta'_n \\ \sigma; \pi_{n+1}(\overline{\theta}) \vdash \mathtt{e}[\mathtt{x}_1 := \rho(\theta'_1), ..., \mathtt{x}_n := \rho(\theta'_n)] \Downarrow \mathtt{v}(\overline{\eta}) \end{array}}{\sigma; \overline{\theta} \vdash \mathtt{d}(\mathtt{e}_1, ..., \mathtt{e}_n) \Downarrow \mathtt{v}(\theta'_1, ..., \theta'_n, \mathtt{v}(\overline{\eta}))}$$

$$\text{[E-SPR]} \frac{\begin{array}{c} \sigma; \pi_0(\overline{\theta}) \vdash \mathtt{e}_0 \Downarrow \eta_0 \quad \cdots \quad \sigma; \pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \eta_n \\ \rho(\eta_0, ..., \eta_n) = \mathtt{v}_0 ... \mathtt{v}_n \qquad \rho(\overline{\theta}) = \mathtt{w}_1 ... \mathtt{w}_m \\ \sigma; \emptyset \vdash \mathtt{f}(\mathtt{w}_1, \mathtt{v}_1, ..., \mathtt{v}_n) \Downarrow \mathtt{u}_1(\cdots) \quad \cdots \quad \sigma; \emptyset \vdash \mathtt{f}(\mathtt{w}_m, \mathtt{v}_1, ..., \mathtt{v}_n) \Downarrow \mathtt{u}_m(\cdots) \end{array}}{\sigma; \overline{\theta} \vdash \{\mathtt{e}_0 : \mathtt{f}(@, \mathtt{e}_1, ..., \mathtt{e}_n)\} \Downarrow \bigwedge\{\mathtt{v}_0, \mathtt{u}_1, ..., \mathtt{u}_m\}(\eta_0, \eta_1, ..., \eta_n)}$$

Figure 6: Big-step operational semantics for expression evaluation

The rules of the operational semantics are *syntax directed*, namely, the rule used for deriving a judgement $\sigma; \overline{\theta} \vdash \mathtt{e} \Downarrow \theta$ is univocally determined by $\mathtt{e}$ (cf. Figure 6). Therefore, the shape of the value-tree θ is univocally determined by $\mathtt{e}$, and the whole value-tree is univocally determined by σ, $\overline{\theta}$, and $\mathtt{e}$.

The rules of the operational semantics are almost standard, with the exception that rules [E-COND], [E-BLT], [E-DEF] and [E-SPR] use the auxiliary function $\pi_i(\cdot)$ to ensure that, in the judgements in the premise of the rule, the value-tree environment is aligned with the expression to be evaluated. Note that the semantics of conditional expressions prescribes that both the branches of the conditional are evaluated.[2]

The most important rule is [E-SPR] which handles spreading expressions formalising the description provided in Section 2.1. It first recursively evaluates expressions $\mathtt{e}_i$ to value-trees η_i (after proper alignment of value-tree environment by operator $\pi_i(.)$) with top-level values $\mathtt{v}_i$. Then it gets from neighbours their values $\mathtt{w}_j$ for the spreading expression, and for each of them $\mathtt{f}$ is evaluated giving top-level result $\mathtt{u}_j$. The resulting value is then obtained by the minimum among $\mathtt{v}_0$ and the

[2]Our calculus does not model the *domain restriction* construct in [46, 36].

values $\mathtt{u}_j$ (which equates to $\mathtt{v}_0$ if there are currently no neighbours). Note that, since in a spreading expression $\{\mathtt{e}_0 : \mathtt{f}(\mathtt{@},\mathtt{e}_1,...,\mathtt{e}_n)\}$ the function f must be a diffusion and diffusions are pure functions (cf. Section 3.1), only the root of the value-tree produced by the evaluation of the application of f to the values of $\mathtt{e}_1, \ldots, \mathtt{e}_n$ must be stored (c.f. the conclusion of rule [E-SPR]). We will in the following provide a network semantics taking care of associating to each device the set of neighbour trees against which it performs a computation round, namely, connecting this operational semantics to the actual network topology.

Example 3.5 (About device semantics)**.** Consider the program **P**:

```
def real main() is { #src :  @ + #dist },
```

where #src and #dist are sensors of type real, and + is the built-in sum operator which has type-signature real(real,real).

The evaluation of $\mathtt{e_{main}} =$ { #src : @ + #dist } on a device ι_1 when

- the current sensor-value map for ι_1 is σ_1 such that $\sigma_1(\mathtt{\#src}) = 0$ and $\sigma_1(\mathtt{\#dist}) = 1$, and
- ι_1 has currently no neighbours,

(expressed by the judgement $\sigma_1;\emptyset \vdash \mathtt{e_{main}} \Downarrow \theta_1$) yields the value-tree $\theta_1 = 0(0,1)$ by rule [E-SPR], since: $n = 1$; the evaluation of $\mathtt{e}_0 = \mathtt{\#src}$ yields $\eta_0 = 0()$ (by rule [E-SNS]); the evaluation of $\mathtt{e}_1 = \mathtt{\#dist}$ yields $\eta_1 = 1()$ (by rule [E-SNS]); $m = 0$; and $\bigwedge\{0\} = 0$.

Similarly, the evaluation of $\mathtt{e_{main}}$ on a device ι_2 when

- the current sensor-value map for ι_2 is σ_2 such that $\sigma_2(\mathtt{\#src}) = 8$ and $\sigma_2(\mathtt{\#dist}) = 1$, and
- ι_2 has currently no neighbours,

(expressed by the judgement $\sigma_2;\emptyset \vdash \mathtt{e_{main}} \Downarrow \theta_2$) yields the value-tree $\theta_2 = 8(8,1)$ by rule [E-SPR], since: $n = 1$; the evaluation of $\mathtt{e}_0 = \mathtt{\#src}$ yields $\eta_0 = 8()$ (by rule [E-SNS]); the evaluation of $\mathtt{e}_1 = \mathtt{\#dist}$ yields $\eta_1 = 1()$ (by rule [E-SNS]); $m = 0$; and $\bigwedge\{8\} = 8$.

Then, the evaluation of $\mathtt{e_{main}}$ on a device ι_3 when

- the current sensor-value map for ι_3 is σ_3 such that $\sigma_3(\mathtt{\#src}) = 4$ and $\sigma_3(\mathtt{\#dist}) = 1$, and
- ι_3 has neighbours ι_1 and ι_2,

(expressed by the judgement $\sigma_3;\theta_1\theta_2 \vdash \mathtt{e_{main}} \Downarrow \theta_3$) yields the value-tree $\theta_3 = 1(4,1)$ by rule [E-SPR], since: $n = 1$; the evaluation of $\mathtt{e}_0 = \mathtt{\#src}$ yields $\eta_0 = 4()$ (by rule [E-SNS]); the evaluation of $\mathtt{e}_1 = \mathtt{\#dist}$ yields $\eta_1 = 1()$ (by rule [E-SNS]); $m = 2$; the evaluation of $0+1$ yields $1(0(),1())$ (by rule [E-BLT]); the evaluation of $8+1$ yields $9(0(),8())$ (by rule [E-BLT]); and $\bigwedge\{4,1,9\} = 1$.

3.2.2. *Network Evolution.* We now provide an operational semantics for the evolution of whole networks, namely, for modelling the distributed evolution of computational fields over time. Figure 7 (top) defines key syntactic elements to this end. F models the overall computational field (state), as a map from device identifiers to value-trees. τ models *network topology*, namely, a directed neighbouring graph, as a map from device identifiers to set of identifiers. Σ models *sensor (distributed) state*, as a map from device identifiers to (local) sensors (i.e., sensor name/value maps). Then, E (a couple of topology and sensor state) models the system's environment. So, a whole network configuration N is a couple of a field and environment.

We define network operational semantics in terms of small-steps transitions of the kind $N \xrightarrow{\ell} N'$, where ℓ is either a device identifier in case it represents its firing, or label ε to model any environment change. This is formalised by the two rules in Figure 7 (bottom). Rule [N-FIR] models a network evolution due to a computation round (firing) at device ι: it reconstructs the proper local environment,

System configurations and action labels:

F	$::=$	$\overline{\iota} \triangleright \overline{\theta}$	computational field
τ	$::=$	$\overline{\iota} \triangleright \overline{I}$	topology
Σ	$::=$	$\overline{\iota} \triangleright \overline{\sigma}$	sensors-map
E	$::=$	τ, Σ	environment
N	$::=$	$\langle E; F \rangle$	network configuration
ℓ	$::=$	$\iota \mid \varepsilon$	action label

Environment well-formedness:

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

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

$$\text{[N-FIR]}\ \frac{E = \tau, \Sigma \qquad \tau(\iota) = \overline{\iota} \qquad \Sigma(\iota); F(\overline{\iota}) \vdash \mathtt{e_{main}} \Downarrow \theta}{\langle E; F \rangle \xrightarrow{\iota} \langle E; F[\iota \triangleright \theta] \rangle}$$

$$\text{[N-ENV]}\ \frac{\begin{array}{l} WFE(E') \qquad E' = \tau, \iota_1 \triangleright \sigma_1, \ldots, \iota_n \triangleright \sigma_n \\ \sigma_1; \emptyset \vdash \mathtt{e_{main}} \Downarrow \theta_1 \quad \cdots \quad \sigma_n; \emptyset \vdash \mathtt{e_{main}} \Downarrow \theta_n \qquad F_0 = \iota_1 \triangleright \theta_1, \ldots, \iota_n \triangleright \theta_n \end{array}}{\langle E; F \rangle \xrightarrow{\varepsilon} \langle E'; F_0[F] \rangle}$$

Figure 7: Small-step operational semantics for network evolution

taking local sensors ($\Sigma(\iota)$) and accessing the value-trees of ι's neighbours;[3] then by the single device semantics we obtain the device's value-tree θ, which is used to update system configuration. Rule [N-ENV] models a network evolution due to change of the environment E to an arbitrarily new well-formed environment E'—note that this encompasses both neighborhood change and addition/removal of devices. Let $\iota_1, \ldots, \iota_n$ be the domain of E'. We first construct a field F_0 associating to all the devices of E' the default value-trees $\theta_1, \ldots, \theta_n$ obtained by making devices perform an evaluation with no neighbours and sensors as of E'. Then, we adapt the existing field F to the new set of devices: $F_0[F]$ automatically handles removal of devices, map of new devices to their default value-tree, and retention of existing value-trees in the other devices.

Example 3.6 (About network evolution)**.** Consider a network of devices running the program **P** of Example 3.5. The initial situation, when (the functionality associated to) the program **P** is switched-off on all the devices, is modelled by the empty network configuration $\langle \emptyset \triangleright \emptyset, \emptyset \triangleright \emptyset; \emptyset \triangleright \emptyset \rangle$.

The network evolution representing the fact that the environment evolves because device ι_3 switches-on when its sensor `#src` perceives value 0 and its sensor `#dist` perceives value 1 (and gets initialised to the value-tree obtained by firing with respect to the empty set of neighbourhs), is modelled (according to rule [N-ENV]) by the reduction step $\langle \emptyset \triangleright \emptyset, \emptyset \triangleright \emptyset; \emptyset \triangleright \emptyset \rangle \xrightarrow{\varepsilon} \langle \iota_3 \triangleright \emptyset, \iota_3 \triangleright \sigma_1; \iota_3 \triangleright \theta_1 \rangle$, where the sensor-value mapping σ_1 and the value-tree θ_1 are those introduced in Example 3.5.

Then, the network evolution representing the fact that the environment evolves is as follows:

- the device ι_1 switches-on when its sensor `#src` perceives value 0 and its sensor `#dist` perceives value 1, and has only ι_3 as neighbour;

[3] The operational semantics abstracts from the details of message broadcast from/to neighbours: the most recent value-trees received by a device ι from its neighbours while it was sleeping are identified with the value-trees associated to the neighbours of the device ι when it fires.

- the device ι_2 switches-on when its sensor `#src` perceives value 4 and its sensor `#dist` perceives value 1, and has no neighbours, and
- on device ι_3 sensor `#src` perceives value 8, sensor `#dist` perceives value 1, and ι_3 has ι_1 and ι_2 as neighbours,

is modelled (according to rule [N-ENV]) by the reduction step $\langle \iota_3 \triangleright \emptyset, \iota_3 \triangleright \sigma_1; \iota_3 \triangleright \theta_1 \rangle \xrightarrow{\varepsilon} \langle \tau, \Sigma; F \rangle$, where

$$\tau = \iota_1 \triangleright \{\iota_3\}, \iota_2 \triangleright \emptyset, \iota_3 \triangleright \{\iota_1, \iota_2\}, \quad \Sigma = \iota_1 \triangleright \sigma_1, \iota_2 \triangleright \sigma_2, \iota_3 \triangleright \sigma_3, \quad F = \iota_1 \triangleright \theta_1, \iota_2 \triangleright \theta_2, \iota_3 \triangleright \theta_1$$

and the sensor-value mapping σ_2, σ_3 and the value-tree θ_2 are those introduced in Example 3.5.

Finally, the network evolution representing the fact that device ι_3 fires is modelled (according to rule [N-FIR]) by the reduction step $\langle \tau, \Sigma; F \rangle \xrightarrow{\iota_3} \langle \tau, \Sigma; \iota_1 \triangleright \theta_1, \iota_2 \triangleright \theta_2, \iota_3 \triangleright \theta_3 \rangle$, where the value-tree θ_2 is that introduced in Example 3.5.

3.3. **The Self-stabilisation Property.** Upon this semantics, we introduce the following definitions and notations ending with the self-stabilisation property.

Initiality: The empty network configuration $\langle \emptyset \triangleright \emptyset, \emptyset \triangleright \emptyset; \emptyset \triangleright \emptyset \rangle$ is said *initial*.

Reachability: Write $N \stackrel{\overline{\ell}}{\Longrightarrow} N'$ as short for $N \xrightarrow{\ell_1} N_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_n} N'$. A configuration N is said *reachable* if $N_0 \stackrel{\overline{\ell}}{\Longrightarrow} N$ where N_0 is initial. Reachable configurations are the well-formed ones, and in the following we shall implicitly consider only reachable configurations.

Firing: A firing evolution from N to N', written $N \Longrightarrow N'$, is one such that $N \stackrel{\overline{\iota}}{\Longrightarrow} N'$ for some $\overline{\iota}$, namely, where only firings occur.

Stability: A network configuration N is said *stable* if $N \xrightarrow{\iota} N'$ implies $N = N'$, namely, the computation of fields reached a fixpoint in the current environment. Note that if N is stable, then it also holds that $N \Longrightarrow N'$ implies $N = N'$.

Fairness: We say that a sequence of device fires is *k-fair* ($k \geq 0$) to mean that, for every h ($1 \leq h \leq k$), the h-th fire of any device is followed by at least $k - h$ fires of all the other devices. Accordingly, a firing evolution $N \stackrel{\overline{\iota}}{\Longrightarrow} N'$ is said *k-fair*, written $N \stackrel{\overline{\iota}}{\Longrightarrow}_k N'$, to mean that $\overline{\iota}$ is k-fair. We also write $N \Longrightarrow_k N'$ if $N \stackrel{\overline{\iota}}{\Longrightarrow}_k N'$ for some $\overline{\iota}$. This notion of fairness will be used to characterise finite firing evolutions in which all devices are given equal chance to fire when all others had.

Strong self-stabilisation: A network configuration $\langle E; F \rangle$ is said to *(strongly) self-stabilise* (simply, self-stabilise, in the following) to $\langle E; F' \rangle$ if there is a $k > 0$ and a field F' such that $\langle E; F \rangle \Longrightarrow_k \langle E; F' \rangle$ implies $\langle E; F' \rangle$ is stable, and F' is univocally determined by E. Self-stability basically amounts to the inevitable reachability of a stable state depending only on environment conditions, through a sufficiently long fair evolution. Hence, the terminology is abused equivalently saying that a program **P** or (equivalently) a field expression $\mathtt{e}_{\mathtt{main}}$ is self-stabilising if for any environment state E there exists a unique stable field F' such that *any* $\langle E; F \rangle$ *self-stabilises* to $\langle E; F' \rangle$.

Self-stability: A network configuration $\langle E; F \rangle$ is said *self-stable* to mean that it is stable and F is univocally determined by E.

Note that our definition of self-stabilisation is actually a stronger version of the standard definition of self-stabilisation as given e.g. in [22]—see more details in Section 10. Instead of simply requiring that we enter a "self-stable set" of states and never escape from it, we require that *(i)* such a set has a single element, and *(ii)* such an element is globally unique, i.e., it does not depend on the initial state. Viewed in the context of an open system, it means that we seek for programs self-stabilising in any environment independently of any intermediate computation state. This is a requirement of key

importance, since it entails that any subexpression of the program can be associated to a final and stable field, reached in finite time by fair evolutions and adapting to the shape of the environment. This acts as the sought bridge between the micro-level (field expression in program code), and the macro-level (expected global outcome).

Example 3.7 (A self-stabilising program)**.** Consider a network of devices running the program **P** of Examples 3.5 and 3.6

```
def real main() is { #src :  @ + #dist },
```

and assume that the sensor `#dist` is guaranteed to always return a positive value (recall that the sensors `#src` and `#dist` have type `real` and the built-in operator `+` has type-signature `real(real,real)`). Then, the operator `+` is guaranteed to be used with type `real(real,pr)`, where `pr` is the type of positive reals (i.e., the *refinement type* [26] that refines type `real` by keeping only the positive values), so that the following conditions are satisfied:

(1) `+` is *monotonic nondecreasing* in its first argument, i.e., for any $\mathtt{v}$, $\mathtt{v}'$ and $\mathtt{v}_2$ of type `pr`: $\mathtt{v} \leq_{\mathtt{real}} \mathtt{v}'$ implies $\mathtt{v}+\mathtt{v}_2 \leq_{\mathtt{real}} \mathtt{v}'+\mathtt{v}_2$;
(2) `+` is *progressive* in its first argument, i.e., for any $\mathtt{v}$ and $\mathtt{v}_2$ of type `pr`: $\mathtt{POSINF} = \mathtt{POSINF}+\mathtt{v}_2$, and $\mathtt{v} \neq \mathtt{POSINF}$ implies $\mathtt{v} <_{\mathtt{real}} \mathtt{v}+\mathtt{v}_2$.

Starting from an initial empty configuration, we move by rule [N-ENV] to a new environment with the following features:

- the domain is formed by $2n$ ($n \geq 1$) devices $\iota_1,...,\iota_n,\iota_{n+1},...,\iota_{2n}$;
- the topology is such that any device ι_i is connected to ι_{i+1} and ι_{i-1} (if they exist);
- sensor `#dist` gives 1 everywhere;
- sensor `#src` gives 0 on the devices ι_i ($1 \leq i \leq n$, briefly referred to as *left devices*) and a value u ($u > n+1$) on the devices ι_j ($n+1 \leq j \leq 2n$, briefly referred to as *right devices*).

Accordingly, the left devices are all assigned to value-tree $0(0,1)$, while the right ones to $u(u,1)$: hence, the resulting field maps left devices to 0 and right devices to 1—remember such evaluations are done assuming nodes are isolated, hence the result is exactly the value of the source expression. With this environment, the firing of a device can only replace the root of a value-tree, making it the minimum of the source expression's value and the successor of neighbour's values. Hence, any firing of a device that is not ι_{n+1} does not change its value-tree. When ι_{n+1} fires instead by rule [N-FIR], its value-tree becomes $1(u,1)$, and it remains so if more firings occur next.

Now, only a firing at ι_{n+2} causes a change: its value-tree becomes $2(u,1)$. Going on this way, it is easy to see that after any n-fair firing sequence the network self-stabilises to the field state where left devices still have value-tree $0(u,1)$, while right devices $\iota_{n+1},\iota_{n+2},\iota_{n+3},...$ have value-trees $1(u,1),2(u,1),3(u,1),...$, respectively. That is, the root of such trees form a hop-count gradient, measuring minimum distance to the source nodes, namely, the left devices.

It can also be shown that any environment change, followed by a sufficiently long firing sequence, makes the system self-stabilise again, possibly to a different field state. For instance, if the two connections of ι_{2n-1} to/from ι_{2n-2} break (assuming $n > 2$), the part of the network excluding ι_{2n-1} and ι_{2n} keeps stable in the same state. The values at ι_{2n-1} and ι_{2n} start raising instead, increasing of 2 alternatively until both reach the initial value-trees $u(u,1)$—and this happens in finite time by a fair evolution thanks to the local noetherianity property of stabilising diffusions. Note that the final state is still the hop-count gradient, though adapted to the new environment topology.

Example 3.8 (A non self-stabilising program)**.** An example of program that is *not* self-stabilising is

```
def real main() is { #src :  id(@) }
```

(written `def real main() is { #src :  @ }` for short). There, the diffusion is the identity function: `real id(real x) is x`, which (under the assumption, done in Example 3.7, that the sensor `#scr` is guaranteed to return positive values) is guaranteed to be used with signature `(pr)pr` and is not progressive in its first argument (c.f. condition (1) in Example 3.7).[4]

Assuming a connected network, and `#src` holding value $\mathtt{v}_s$ in one node and `POSINF` in all others, then *any* configuration where all nodes hold the same value `v` less than or equal to $\mathtt{v}_s$ is trivially stable. This would model a source gossiping a fixed value $\mathtt{v}_s$ everywhere: if the source suddenly gossips a value $\mathtt{v}'_s$ smaller than `v`, then the network would self-organise and all nodes would eventually hold $\mathtt{v}'_s$. However, if the source then gossips a value $\mathtt{v}''_s$ greater than $\mathtt{v}'_s$, the network would *not* self-organise and all nodes would remain stuck to value $\mathtt{v}'_s$.

4. Sorts, Stabilising Diffusions and the Stabilising-Diffusion Condition

In this section we state a sufficient condition for self-stabilisation. This condition is about the behaviour of a diffusion (cf. Definition 3.2) on a subsets of its arguments (cf. Example 3.7). We first introduce *refinement types* (or *sorts*) as a convenient way to denote these subsets (in Section 4.1) and then use them to formulate the notion of *stabilising diffusion* (in Section 4.2) and the sufficient condition for self-stabilisation (in Section 4.3).

4.1. Refinement Types (or Sorts).

Refinement types [26] provide a mean to conservatively extend the static type system of a language by providing the ability of specify *refinements* of each type. All the programs accepted by the original type system are accepted by the refinement-type system and vice-versa, but refinement-types provide additional information that support stating and checking properties of programs. Following [17], we refer to refinement types as *sorts* and use terms like *subsorting* and *sort checking*.

For instance, for the ground type `real` of reals we consider the six *ground sorts* `nr` (negative reals), `zr` (the sort for `0`), `pr` (positive reals), `znr` (zero or negative reals), `zpr` (zero or positive reals), and `real` (each type trivially refines itself); while for the type `bool` we consider the thee sorts `false` (the sort for `FALSE`), `true` (the sort for `TRUE`) and `bool`. Each sort-signature has the same structure of the type-signature it refines. For instance, we can build $9(=3^2)$ *sort-signatures* for the type-signature `bool(bool)`:

```
false(false),  false(true),  false(bool),
true(false),   true(true),   true(bool),
bool(false),   bool(true),   bool(bool).
```

We assume a mapping $\textbf{sorts}(\cdot)$ that associates to each type the (set of) sorts that refine it, and a mapping $\textbf{sort-signatures}(\cdot)$ that associates to each type-signature the (set of) sort-signatures that refine it (note that the latter mapping is determined by the the former, i.e., by the value of $\textbf{sorts}(\cdot)$ on ground types). A type `T` trivially refines itself, i.e., for every type `T` it holds that $\mathtt{T} \in \textbf{sorts}(\mathtt{T})$. Similarly, for every type-signature $\mathtt{T}(\overline{\mathtt{T}})$ it holds that $\mathtt{T}(\overline{\mathtt{T}}) \in \textbf{sort-signatures}(\mathtt{T}(\overline{\mathtt{T}}))$. We write $[\![\mathtt{S}]\!]$ to denote the set of values of sort `S`. Note that, by construction:

$$\text{for all } \mathtt{S} \in \textbf{sorts}(\mathtt{T}) \text{ it holds that } [\![\mathtt{S}]\!] \subseteq [\![\mathtt{T}]\!].$$

Sorts and sort-signatures express properties of expressions and functions, respectively. We say that:

[4] The function `id` is progressive whenever its is used with a signature of the form `(real_n)real_n`, where `real_n` is the refinement type that refines `real_n` by keeping only the value `n`—in the example, this corresponds to the case when the sensor `#scr` is guaranteed to always return the constant value `n` on all the devices.

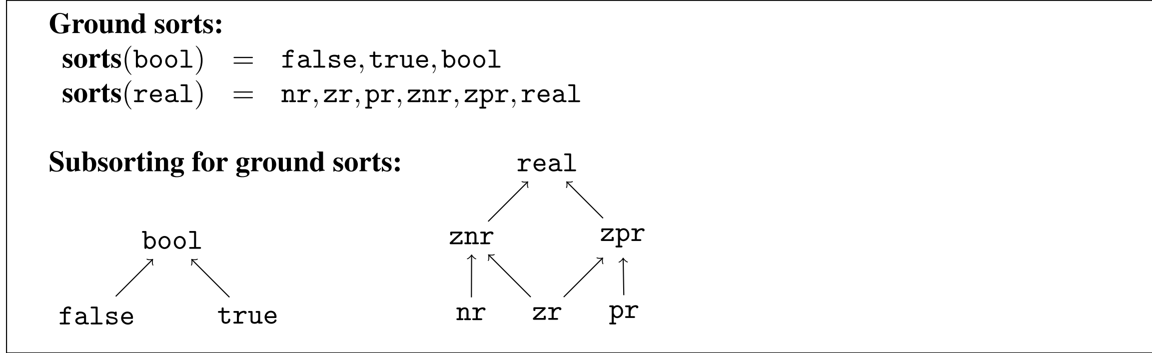

Figure 8: Sorts and subsorting for the ground types used in the examples

- a value $\mathtt{v}$ *has* (or *satisfies*) sort $\mathtt{S}$ to mean that $\mathtt{v} \in [\![\mathtt{S}]\!]$ holds—we write **sorts**$(\mathtt{v})$ to denote the set of all the sorts satisfied by $\mathtt{v}$, and
- a pure function $\mathtt{f}$ *has* (or *satisfies*) sort-signature $\mathtt{S}(\overline{\mathtt{S}})$ to mean that for all $\overline{\mathtt{v}} \in [\![\overline{\mathtt{S}}]\!]$ it holds that $[\![\mathtt{f}]\!](\overline{\mathtt{v}}) \in [\![\mathtt{S}]\!]$—we write **sort-signatures**$(\mathtt{f})$ to denote the set of all the sort-signatures satisfied by $\mathtt{f}$.

For every sort $\mathtt{S}$ in **sorts**$(\mathtt{T})$ we write $\leq_{\mathtt{S}}$ to denote the restriction to $[\![\mathtt{S}]\!]$ of the total order $\leq_{\mathtt{T}}$ (cf. Section 2.1) and write $\top_{\mathtt{S}}$ to denote the maximum element of $[\![\mathtt{S}]\!]$ with respect to $\leq_{\mathtt{S}}$.

Subsorting is a partial order relation over sorts of each type that models the inclusion relationship between them. For instance, each positive number is a zero or positive number, so will write $\mathtt{nr} \leq \mathtt{znr}$ to indicate that $\mathtt{nr}$ is a subsort of $\mathtt{znr}$. We require that the subsorting relation satisfies the following (fairly standard) conditions:

(1) The type $\mathtt{T}$ is the maximum element with respect to subsorting relation on **sorts**$(\mathtt{T})$.
(2) For every $\mathtt{S}_1, \mathtt{S}_2 \in$ **sorts**$(\mathtt{T})$ there exists a least upper bound in **sorts**$(\mathtt{T})$—that we denote by $sup_{\leq}(\mathtt{S}_1, \mathtt{S}_2)$.
(3) For each value $\mathtt{v}$ the set of sorts **sorts**$(type(\mathtt{v}))$ has a minimum element w.r.t. $\leq$.
(4) The subsorting relation is *sound* and *complete* according to the semantics of sorts, i.e.,

$$\mathtt{S} \leq \mathtt{S}' \text{ if and only if } [\![\mathtt{S}]\!] \subseteq [\![\mathtt{S}']\!].$$

Example 4.1. Figure 8 illustrates the sorts and the subsorting for the ground types used in the examples introduced throughout the paper.

Subsigning is the partial order obtained by lifting subsorting to sort-signatures by the following subsigning rule (which, as usual, is covariant in the result sort and contravariant in the argument sorts):

$$\text{[I-SIG]}\ \frac{\mathtt{S} \leq \mathtt{S}' \quad \overline{\mathtt{S}}' \leq \overline{\mathtt{S}}}{\mathtt{S}(\overline{\mathtt{S}}) \leq \mathtt{S}'(\overline{\mathtt{S}'})}$$

According to the above explanations, for every type $\mathtt{T}$ and type-signature $\mathtt{T}(\overline{\mathtt{T}})$ we have that both $($**sorts**$(\mathtt{T}), \leq)$ and $($**sort-signatures**$(\mathtt{T}(\overline{\mathtt{T}})), \leq)$ are partial orders.

4.2. **Stabilising Diffusions.** Recall the notion of diffusion (Definition 3.2). In this section we exploit sorts to formulate the notion of *stabilising diffusion*: a predicate on the behaviour of a diffusion that will be used to express the sufficient condition for self-stabilisation. The stabilising diffusion predicate specifies constraints on the behaviour of a diffusion $\mathtt{f}$ of type $\mathtt{T}_1(\mathtt{T}_1, ..., \mathtt{T}_n)$ by exploiting a sort-signature $\mathtt{S}(\mathtt{S}_1, ..., \mathtt{S}_n) \in$ **sort-signatures**$(\mathtt{f})$.

Definition 4.2 (Stabilising diffusion)**.** A diffusion $\mathtt{f}$ is *stabilising with respect to the sort-signature* $\mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}}) \in$ **sort-signatures**$(\mathtt{f})$ such that $\mathtt{S} \leq \mathtt{S}_1$ (notation *stabilising*$(\mathtt{f},\mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}})))$ if the following conditions hold:

(1) $\mathtt{f}$ is *monotonic nondecreasing* in its first argument, i.e., for all $\mathtt{v} \in [\![\mathtt{S}_1]\!]$, $\mathtt{v}' \in [\![\mathtt{S}_1]\!]$ and $\overline{\mathtt{v}} \in [\![\overline{\mathtt{S}}]\!]$: $\mathtt{v} \leq_{\mathtt{S}_1} \mathtt{v}'$ implies $[\![\mathtt{f}]\!](\mathtt{v},\overline{\mathtt{v}}) \leq_{\mathtt{S}_1} [\![\mathtt{f}]\!](\mathtt{v}',\overline{\mathtt{v}})$;

(2) $\mathtt{f}$ is *progressive* in its first argument, i.e., for all $\overline{\mathtt{v}} \in [\![\overline{\mathtt{S}}]\!]$: $[\![\mathtt{f}]\!](\top_{\mathtt{S}_1},\overline{\mathtt{v}}) =_{\mathtt{S}_1} \top_{\mathtt{S}_1}$ and, for all $\mathtt{v} \in [\![\mathtt{S}_1]\!] - \{\top_{\mathtt{S}_1}\}$, $\mathtt{v} <_{\mathtt{S}_1} [\![\mathtt{f}]\!](\mathtt{v},\overline{\mathtt{v}})$.

We say that the sort-signature $\mathtt{S}(\overline{\mathtt{S}})$ is *stabilising for* $\mathtt{f}$ to mean that *stabilising*$(\mathtt{f},\mathtt{S}(\overline{\mathtt{S}}))$ holds, and write **stabilising-sort-signatures**$(\mathtt{f})$ to denote set of the stabilising sort-signatures for $\mathtt{f}$.

Example 4.3. Consider the library in Figure 2. The following predicates hold:

- *stabilising*$(\mathtt{+},\mathtt{zr}(\mathtt{zr},\mathtt{zr}))$,
- *stabilising*$(\mathtt{+},\mathtt{pr}(\mathtt{zpr},\mathtt{pr}))$,
- *stabilising*$(\mathtt{+},\mathtt{real}(\mathtt{real},\mathtt{pr}))$, and
- *stabilising*$(\mathtt{restrictSum},\mathtt{real}(\mathtt{real},\mathtt{pr},\mathtt{bool}))$.

Note that Condition (2) in Definition 4.2 introduces a further constraint between the sort of the first argument $\mathtt{S}_1$ and the sort of the result $\mathtt{S}$ in the sort-signature $\mathtt{S}(\mathtt{S}_1,...,\mathtt{S}_n)$ used for $\mathtt{f}$. For instance, given the diffusion

```
def real f(real x, real y) is -(x+y)
```

the sort signature $\mathtt{nr}(\mathtt{pr},\mathtt{pr}) \in$ **sort-signatures**$(\mathtt{f}) \subset$ **sort-signatures**$(\mathtt{real}(\mathtt{real},\mathtt{real}))$ is not compatible with Condition (2), since $\mathtt{v}_1,\mathtt{v}_2 \in [\![\mathtt{pr},\mathtt{pr}]\!]$ and $\mathtt{v} = [\![\mathtt{f}]\!](\mathtt{v}_1,\mathtt{v}_2) \in \mathtt{nr}$ imply $\mathtt{v}_1 \not\leq \mathtt{v}$. Namely, the sort of the result $\mathtt{S}$ and the sort of the first argument $\mathtt{S}_1$ must be such that $[\![\mathtt{S}]\!] \subseteq^{\text{progressive}} [\![\mathtt{S}_1]\!]$, where the relation $\subseteq^{\text{progressive}}$ between two subsets $\mathbf{S}$ and $\mathbf{S}_1$ of $[\![\mathtt{T}]\!]$ (i.e., between elements of the powerset $\mathscr{P}([\![\mathtt{T}]\!])$), that we call *progressive inclusion*, is defined as follows:

$$\mathbf{S} \subseteq^{\text{progressive}} \mathbf{S}_1 \quad \text{if and only if} \quad \mathbf{S} \subseteq \mathbf{S}_1 \quad \text{and} \quad \top_{\mathtt{S}} = \top_{\mathtt{S}_1}.$$

We write $\leq^{\text{progressive}}$ to denote the *progressive subsorting* relation, which is the restriction of subsorting relation defined as follows:

$$\mathtt{S} \leq^{\text{progressive}} \mathtt{S}' \text{ if and only if } [\![\mathtt{S}]\!] \subseteq^{\text{progressive}} [\![\mathtt{S}']\!].$$

To summarise: if a sort signature $\mathtt{S}(\overline{\mathtt{S}})$ is stabilising for some diffusion, then $\mathtt{S}(\overline{\mathtt{S}})$ must be *progressive*, according to the following definition.

Definition 4.4 (Progressive sort-signature)**.** A sort-signature $\mathtt{S}(\overline{\mathtt{S}})$ with $\overline{\mathtt{S}} = \mathtt{S}_1,...,\mathtt{S}_n$ $(n \geq 1)$ is a *progressive sort-signature* (notation *progressive*$(\mathtt{S}(\overline{\mathtt{S}})))$ if $\mathtt{S} \leq^{\text{progressive}} \mathtt{S}_1$.

Given a diffusion type-signature $\mathtt{T}(\overline{\mathtt{T}})$ (cf. Definition 3.2) we write **progressive-sort-signatures**$(\mathtt{T}(\overline{\mathtt{T}}))$ to denote the (set of) progressive sort-signatures that refine it.

Example 4.5. Figure 9 illustrates the progressive subsorting for the ground sorts used in the examples introduced throughout the paper.

The following partial order between progressive sort-signatures, that we call *stabilising subsigning*:

$$\text{[I-S-SIG]}\ \frac{\mathtt{S} \leq^{\text{progressive}} \mathtt{S}' \quad \mathtt{S}'_1 \leq^{\text{progressive}} \mathtt{S}_1 \quad \overline{\mathtt{S}}' \leq \overline{\mathtt{S}}}{\mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}}) \leq^{\text{stabilising}} \mathtt{S}'(\mathtt{S}'_1\overline{\mathtt{S}}')}$$

captures the natural implication relation between stabilisation properties, as stated by the following proposition.

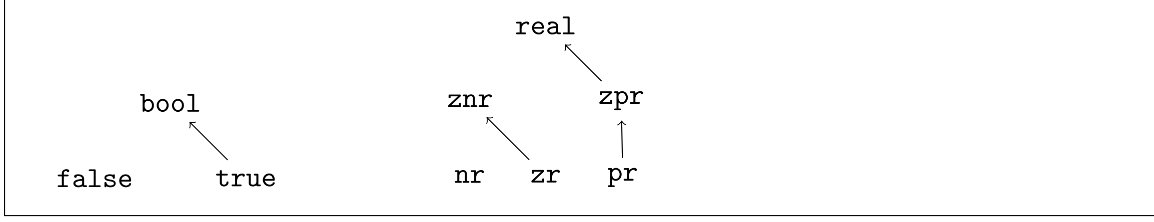

Figure 9: Progressive subsorting for the ground sorts used in the examples (cf. Figure 8)

Proposition 4.6 (Soudness of stabilising subsigning)**.** *If* $stabilising(f, S(\overline{S}))$ *and* $S(\overline{S}) \leq^{stabilising} S'(\overline{S'})$*, then* $stabilising(f, S'(\overline{S'}))$*.*

Proof. Straightforward from Definition 4.2 and the definition of progressive subsigning (Rule [I-S-SIG] above). □

4.3. **The Stabilising-Diffusion Condition.** We have now all the ingredients to formulate a sufficient condition for self-stabilisation of a well-typed program **P**, that we call the *stabilising-diffusion condition*: any diffusion f used in a spreading expression e of the program (or library) **P** must be a stabilising diffusion with respect a sort-signature for f that correctly describes the sorts of the arguments of f in e. More formally, a well-typed program (or library) **P** satisfies the stabilising-diffusion condition if and only if it admits *valid sort-signature and stabilising assumptions for diffusions*. I.e., for each diffusion-expression $\{e_1 : f(@, e_2, ..., e_n)\}$ occurring in **P**, there exists a sort-signature $S(S_1, \ldots, S_n)$ such that the following two conditions are satisfied.

(1) **Validity of the sort-signature assumption:** the diffusion f has sort-signature $S(S_1, \ldots, S_n)$ and, in any reachable network configuration, the evaluation of the subexpression e_i yields a value $v_i \in [\![S_i]\!]$ $(1 \leq i \leq n)$.
(2) **Validity of the stabilising assumption:** the sort-signature $S(S_1, \ldots, S_n)$ is stabilising for the diffusion f.

Example 4.7. In the body of the function main in Example 3.7, which defines a self-stabilising field:

(1) the diffusion function + is applied according to the sort-signature real(real,pr) since its second argument is the sensor #dist of type real that is guaranteed to always return a value of sort pr; and
(2) *stabilising*(+,real(real,pr)) holds (cf. Example 4.3).

Therefore, the stabilising diffusion condition is satisfied. Also the library in Figure 2 satisfies the stabilising diffusion condition.

Instead, for the diffusion function id of type-signature real(real) used in the non-self-stabilising spreading expression considered in Example 3.8, $S(S_1) = zr(zr)$ is the only sort-signature such that *stabilising*$(id, S(S_1))$ holds.[5] Therefore, since the sensor #src returns a value of sort pr, the stabilising diffusion condition cannot be satisfied.

Remark 4.8 (On choosing the refinements of type real)**.** The choice of the refinements for type real that we considered in the examples is somehow arbitrary. We have chosen a set of refinements that is expressive enough in order the show that all the self-stabilising examples considered in the paper satisfy the stabilising-diffusion condition. For instance, dropping the refinements nr and

[5] Considering the sorts given in Figure 8—c.f. the footnote in Example 3.8.

pr would make it impossible to show that the program considered in Example 3.7 satisfies the stabilising-diffusion condition.

Considering a richer set of refinement would allow to prove more properties of programs (and would make the check more complex). For instance, adding the refinement npr (negative or positive number) such that:

$$\texttt{nr} \leq \texttt{npr} \qquad \texttt{pr} \leq \texttt{npr} \qquad \texttt{npr} \leq \texttt{real}$$

would allow to assign (according to the sort-checking rules presented in Section 8) sort npr to the expressions (x ? -1 : 1), to assume sort-signature $\texttt{false}(\texttt{zr},\texttt{npr})$ for the built-in equality operator on reals $=$, and therefore to check that the user-defined function

```
def <bool> f(<bool> x) is 0 = (x ? -1 : 1)
```

has sort-signature $\texttt{false}(\texttt{bool})$. Although, this would allow to show that more program satisfies the stabilising-diffusion, the refinement npr is not needed in order to show that the self-stabilising examples considered in the paper satisfy the stabilising-diffusion condition. Therefore we have not considered it in the examples.

5. Programs that Satisfy the Stabilising-Diffusion Condition Self-stabilise

In this section we prove the main properties of the proposed calculus, namely: type soundness and termination of device computation (in Section 5.1), and self-stabilisation of network evolution for programs that satisfy the stabilising-diffusion condition (in Section 5.2).

As already mentioned, our notion of self-stabilisation is key as it allows one to conceptually map any (self-stabilising) field expression to its final and stable field state, reached in finite time by fair evolutions and adapting to the shape of the environment. This acts as the sought bridge between the micro-level (field expression in program code), and the macro-level (expected global outcome). In order facilitate the exploitation of this bridge it would be useful to have an effective means for checking the stabilising-diffusion condition. A technique providing such an effective means (for the extension of the calculus with pairs introduced in Section 6) is illustrated in Sections 7, 8 and 9.

5.1. Type Soundness and Termination of Device Computation.

In order to state the properties of device computation we introduce the notion of set of well-typed values trees for an expression.

Given an expression e such that $\overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$, the set $\mathit{WTVT}(\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{T})$ of the *well-typed* value-trees for e, is inductively defined as follows: $\theta \in \mathit{WTVT}(\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{T})$ if there exist

- a sensor mapping σ;
- well-formed tree environments $\overline{\theta} \in \mathit{WTVT}(\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{T})$; and
- values $\overline{\mathtt{v}}$ such that $\mathit{length}(\overline{\mathtt{v}}) = \mathit{length}(\overline{\mathtt{x}})$ and $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{T}}$;

such that $\sigma;\overline{\theta} \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] \Downarrow \theta$ holds—note that this definition is inductive, since the sequence of evaluation trees $\overline{\theta}$ may be empty.

As this notion is defined we can state the following two theorems, guaranteeing that from a properly typed environment, evaluation of a well-typed expression yields a properly typed result and always terminates, respectively.

Theorem 5.1 (Device computation type preservation)**.** *If $\overline{x} : \overline{T} \vdash e : T$, σ is a sensor mapping, $\overline{\theta} \in WTVT(\overline{x} : \overline{T}, e, T)$, $length(\overline{v}) = length(\overline{x})$, $\emptyset \vdash \overline{v} : \overline{T}$ and $\sigma;\overline{\theta} \vdash e[\overline{x} := \overline{v}] \Downarrow \theta$, then $\emptyset \vdash \rho(\theta) : T$.*

Proof. See Appendix A. □

Theorem 5.2 (Device computation termination)**.** *If* $\overline{x}:\overline{T}\vdash \mathsf{e}:T$*,* σ *is a sensor mapping,* $\overline{\theta}\in \mathit{WTVT}(\overline{x}:\overline{T},\mathsf{e},T)$*,* $\mathit{length}(\overline{v})=\mathit{length}(\overline{x})$ *and* $\emptyset\vdash\overline{v}:\overline{T}$*, then* $\sigma;\overline{\theta}\vdash \mathsf{e}[\overline{x}:=\overline{v}]\Downarrow\theta$ *for some value-tree* θ*.*

Proof. See Appendix A. □

The two theorems above guarantee type-soundness and termination of device computations, that is: the evaluation of a well-typed program on any device completes without type errors assuming that the values received from the sensors and the values in the value trees received from the neighbours are well-typed.

5.2. **Self-stabilisation of Network Evolution for Programs that Satisfy the Stabilising-Diffusion Condition.** On top of the type soundness and termination result for device computation we can prove the main technical result of the paper: self-stabilisation of any program that satisfies the stabilising-diffusion condition.

Theorem 5.3 (Network self-stabilisation for programs that satisfy the stabilising-diffusion condition)**.** *Given a program with valid sort and stabilising diffusion assumptions, every reachable network configuration* N *self-stabilises, i.e., there exists* $k\geq 0$ *such that* $N\Longrightarrow_k N'$ *implies that* N' *is self-stable.*

Proof. See Appendix B. □

We conclude this section by giving an outline of the proof of Theorem 5.3. To this aim we first introduce some auxiliary definitions.

Auxiliary Definitions. In the following we omit the subscript $\mathtt{S}$ in $<_{\mathtt{S}}$ and $\leq_{\mathtt{S}}$ when it is clear from the context (i.e., we just write $<$ and $\leq$).

Given a network N with main expression e, we write $\theta_{\iota(\text{in }N)}$ to denote the value-tree of e on device ι in the network configuration N, and write $\mathtt{v}_{\iota(\text{in }N)}$ to denote the value $\rho(\theta_{\iota(\text{in }N)})$ of e on device ι in the network configuration N. Moreover, when $\mathsf{e}=\{\mathsf{e}_0:\mathtt{f}(\mathtt{@},\mathsf{e}_1,\ldots,\mathsf{e}_n)\}$ we write $\theta_{j,\iota(\text{in }N)}$ and write $\mathtt{v}_{j,\iota(\text{in }N)}$ to denote the value-tree and the value of e_j $(0\leq j\leq n)$, respectively. In the following we omit to specify the network configuration N when is is clear from the context, i.e., we simply write θ_ι, $\mathtt{v}_\iota$, $\theta_{j,\iota}$ and $\mathtt{v}_{j,\iota}$.

We say that a device ι is *stable* in N to mean that $N\Longrightarrow N'$ implies $\theta_{\iota(\text{in }N)}=\theta_{\iota(\text{in }N')}$. Note that the following three statements are equivalent:

- N is stable.
- All the devices of N are stable.
- $N\Longrightarrow N'$ implies $N=N'$.

We write *environment*(N) to denote the environment E of a network configuration $N=\langle E;F\rangle$. We say that a device ι is *self-stable* in a network N to mean that it is stable and its value is univocally determined by *environment*(N). Note that a network is self-stable if and only if all its devices are self-stable.

We say that a network N with main expression $\mathsf{e}=\{\mathsf{e}_0:\mathtt{f}(\mathtt{@},\mathsf{e}_1,\ldots,\mathsf{e}_n)\}$ is *pre-stable* to mean that for every device ι in N:

(1) the subexpressions e_i $(0\leq i\leq n)$ are stable, and
(2) $\mathtt{v}_\iota\leq\mathtt{v}_{0,\iota}$.

We say that the network N is *pre-self-stable* to mean that it is pre-stable and the value trees of the subexpressions $\mathtt{e}_i$ ($0 \leq i \leq n$) are self-stable (i.e, they are univocally determined by *environment*(N)). Note that pre-stability is preserved by firing evolution (i.e., if N is pre-stable and $N \Longrightarrow N'$, then N' is pre-stable).

An Outline of the Proof of Theorem 5.3. The proof is by induction on the syntax of closed expressions $\mathtt{e}$ and on the number of function calls that may be encountered during the evaluation of $\mathtt{e}$. Let $\mathtt{e}$ be the main expression of N and $E = environment(N)$. The only interesting case is when $\mathtt{e}$ is a spreading expression $\{\mathtt{e}_0 : \mathtt{f}(@, \mathtt{e}_1, \ldots, \mathtt{e}_n)\}$. By induction there exists $h \geq 0$ such that if $N \Longrightarrow_h N_1$ then on every device ι, the evaluation of $\mathtt{e}_0, \mathtt{e}_1, \ldots, \mathtt{e}_n$ produce stable value-trees $\theta_{0,\iota}, \theta_{1,\iota}, \ldots, \theta_{n,\iota}$, which are univocally determined by E. Note that, if $N \Longrightarrow_{h+1} N_2$ then N_2 is pre-self-stable. Therefore we focus on the case when N is pre-self-stable. The proof of this case is based on the following auxiliary results (which corresponds to Lemmas B.1-B.6 of Appendix B).

B.1 (Minimum value): Any 1-fair evolution $N \Longrightarrow_1 N'$ increases the value of any not self-stable device ι in N such that $\mathtt{v}_{\iota(\text{in } N)}$ is minimum (among the values of the devices in N). The new value $\mathtt{v}_{\iota(\text{in } N')}$ is such that there exists a value $\mathtt{v}'$ such that $\mathtt{v}_{\iota(\text{in } N)} < \mathtt{v}' \leq \mathtt{v}_{\iota(\text{in } N')}$ and in any subsequent firing evolution $N' \Longrightarrow N''$ the value of the device ι will be always greater or equal to $\mathtt{v}'$ (i.e., $\mathtt{v}' \leq \mathtt{v}_{\iota(\text{in } N'')}$).

B.2 (Self-stabilisation of the minimum value): Let $\mathbf{S}_1$ be the subset of the devices in N such that $\mathtt{v}_{0,\iota}$ is minimum (among the values of $\mathtt{e}_0$ in the devices in N). There exists $k \geq 0$ such that any k-fair evolution $N \Longrightarrow_k N'$ is such that

(1) each device ι in $\mathbf{S}_1$ is self-stable in N'.

(2) in N' each device not in $\mathbf{S}_1$ has a value greater or equal then the values of the devices in $\mathbf{S}_1$ and, during any firing evolution, it will always assume values greater than the values of the devices in $\mathbf{S}_1$.

B.3 (Frontier): Let $\mathbf{D}$ be a set of devices. Given a set of stable devices $\mathbf{S} \subseteq \mathbf{D}$ we wrote $frontier_{\mathbf{S}}(\mathbf{D})$ to denote the subset of the devices $\iota \in \mathbf{D} - \mathbf{S}$ such that there exists a device $\iota' \in \mathbf{S}$ such that ι' is a neighbour of ι. If $\mathbf{D}$ are devices of the network and $\mathbf{S}$ satisfies the following conditions

(i) the condition obtained form condition (1) above by replacing $\mathbf{S}_1$ with $\mathbf{S}$,

(ii) the condition obtained form condition (2) above by replacing $\mathbf{S}_1$ with $\mathbf{S}$, and

(iii) $frontier_{\mathbf{S}}(\mathbf{D}) \neq \emptyset$,

then any 1-fair evolution makes the devices in $frontier_{\mathbf{S}}(\mathbf{D})$ self-stable.

B.4 (Minimum value not in S): If $\mathbf{D}$ are devices of the network and $\mathbf{S}$ satisfies conditions *(i)-(iii)* above, satisfies the following condition

(iv) each device in $frontier_{\mathbf{S}}(\mathbf{D})$ is self-stable in N, and

$\mathbf{M} \subseteq \mathbf{D} - \mathbf{S}$ is the set of devices ι such that $\mathtt{v}_{\iota(\text{in } N)}$ is minimum (among the values of the devices in $\mathbf{D} - \mathbf{S}$), and $\mathbf{M} \cap frontier_{\mathbf{S}}(\mathbf{D}) = \emptyset$, then any 1-fair evolution $N \Longrightarrow_1 N'$ increases the value of any not self-stable device ι in $\mathbf{M}$. The new value $\mathtt{v}_{\iota(\text{in } N')}$ is such that there exists a value $\mathtt{v}'$ such that $\mathtt{v}_{\iota(\text{in } N)} < \mathtt{v}' \leq \mathtt{v}_{\iota(\text{in } N')}$ and in any subsequent firing evolution $N' \Longrightarrow N''$ the value of the device ι will be always greater or equal to $\mathtt{v}'$ (i.e., $\mathtt{v}' \leq \mathtt{v}_{\iota(\text{in } N'')}$).

B.5 (Self-stabilisation of the minimum value not in S): If $\mathbf{D}$ are devices of the network and $\mathbf{S}$ satisfies conditions *(i)-(iv)* above, and $\mathbf{M} \subseteq \mathbf{D} - \mathbf{S}$ is the set of devices ι such that $\mathtt{v}_{\iota(\text{in } N)}$ is minimum (among the values of the devices in $\mathbf{D} - \mathbf{S}$), then there exists $k \geq 0$ such that any k-fair evolution $N \Longrightarrow_k N'$ is such that there exists a device ι_1 in $\mathbf{D} - \mathbf{S}$ such that $\mathbf{S}_1 = \mathbf{S} \cup \{\iota_1\}$ satisfies the conditions (1) and (2) above.

T	::=	G | <T,T>	type
e	::=	··· | <e,e> | fst e | snd e	expression

Figure 10: Extensions to the syntax of types and expressions (in Fig. 1) to model pairs

B.6 (Pre-self-stable network self-stabilization): For every reachable pre-self-stable network configuration N there exists $k \geq 0$ such that $N \Longrightarrow_k N'$ implies that N' is self-stable. This sixth auxiliary result, which concludes our proof outline, follows from the previous five auxiliary results. The idea is to consider the auxiliary results B.2, B.3 and B.5 as reasoning steps that may be iterated. We start by applying the auxiliary result B.2 to produce a non-empty set of devices $\mathbf{S}_1$ that satisfies conditions (1) and (2) above. Then, we rename $\mathbf{S}_1$ to $\mathbf{S}$ and iterate the following two reasoning steps until the set of devices $\mathbf{S}$ is such that $\mathit{frontier}_{\mathbf{S}}(\mathbf{D}) = \emptyset$:

- apply the auxiliary results B.3 and B.5 to produce a non-empty set of devices $\mathbf{S}_1$ that satisfies conditions (1) and (2) above; and
- rename $\mathbf{S}_1$ to $\mathbf{S}$.

Clearly the number of iterations is finite (note that $\mathbf{S} = \mathbf{D}$ implies $\mathit{frontier}_{\mathbf{S}}(\mathbf{D}) = \emptyset$). If $\mathbf{S} = \mathbf{D}$ we have done. Otherwise note that, since $\mathit{frontier}_{\mathbf{S}}(\mathbf{D}) = \emptyset$, the evolution of the devices in $\mathbf{D} - \mathbf{S}$ is independent from the devices in $\mathbf{S}$. Therefore we we can iterate the whole reasoning (i.e., starting from the auxiliary result B.2) on the the portion of the network with devices in $\mathbf{D} - \mathbf{S}$.

6. Extending the Calculus with Pairs

The calculus presented in Sections 2 and 3 does not model data structures. In this section we point out that the definitions and results presented in Sections 4 and 5 are parametric in the set of modeled types by considering an extension of the calculus that models the pair data structure. Recent works, such as [8], show that the ability of modelling pairs of values is key in realising context-dependent propagation of information, where along with distance one let the computational field carry additional information as gathered during propagation. In fact, pairs are needed for programming further interesting examples (illustrated in Section 6.2), and their introduction here is useful to better grasp the subtleties of our checking algorithm for self-stabilisation, as described in next sections.

6.1. **Syntax.** The extensions to the syntax of the calculus for modeling pairs are reported in Figure 10. Now types include *pair types* (like `<real,bool>`, `<real,<bool,real>>`,... and so on), expressions include *pair construction* (`<e,e>`) and *pair deconstruction* (`fst e` or `snd e`), and values includes *pair values* (`<1,TRUE>`, `<2,3.5>`, `<<1,FALSE>,3>`,... and so on).

The ordering for ground types has to be somehow lifted to non-ground types. A naural choice for pairs is the lexicographic preorder, i.e, to define $<\mathtt{v}_1,\mathtt{v}_2> \leq_{<\mathtt{T}_1,\mathtt{T}_2>} <\mathtt{v}'_1,\mathtt{v}'_2>$ if either $\mathtt{v}_1 <_{\mathtt{T}_1} \mathtt{v}'_1$ holds or both $\mathtt{v}_1 = \mathtt{v}'_1$ and $\mathtt{v}_2 \leq_{\mathtt{T}_2} \mathtt{v}'_2$ hold.

6.2. **Examples.** In this section we build on the examples introduced in Section 2.2 and show how pairs can be used to program further kinds of behaviour useful in self-organisation mechanisms in general, and also in the specific case of crowd steering, as reported in Figure 11.

The fourth function in Figure 11, called `sector`, can be used to keep track of specific situations during the propagation process. It takes a zero-field source `i` (a field holding value 0 in a “source”, as usual) and a boolean field `c` denoting an area of interest: it creates a gradient of pairs, orderly

```
def <real,bool> sum_or(<real,bool> x, <real,bool> y) is <fst x + fst y, snd x or snd y>
def <real,bool> pt_POSINF_TRUE(<real, bool> x) is ((fst x)=POSINF) ? <POSINF,TRUE> :  x
def <real,bool> sd_sum_or(<real,bool> x, <real,bool> y) is pt_POSINF_TRUE(sum_or(x,y))
def bool sector(real i, bool c) is snd { <i, c> :  sum_or(@,<#dist,c>) }
```

```
def <real,real> add_to_1st(<real,real> x, real y) is <fst x + y, snd x>
def <real,real> pt_POSINF_POSINF(<real, real> x) is ((fst x)=POSINF) ? <POSINF,POSINF> :  x
def <real,real> sd_add_to_1st(<real,real> x, real y) is pt_POSINF_POSINF(add_to_1st(x,y))
def <real,real> gradcast(real i, real j) is { <i, j> :  add_to_1st(@, #dist) }
```

```
def real dist(real i, real j) is gradcast(restrict(j,j==0),grad(i))
def bool path(real i, real j, real w) is (grad(i)+grad(j))+(-w) < dist(i, j)
def real channel(real i, real j, real w) is gradobs(grad(j),not path(i, j, w))
```

Figure 11: Definitions of examples using pairs (see Fig. 2 for the definitions of functions `restrict`, `grad` and `gradobs`)

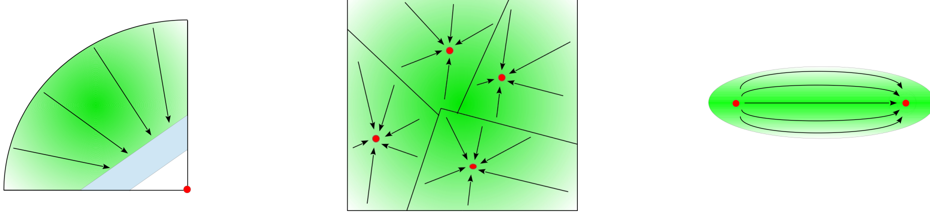

Figure 12: Pictorial representation of sector field (left), partition (center) and channel field (right)

holding distance from source and a boolean value representing whether the route towards the source crossed area `c`. As one such gradient is produced, it is wholly applied to operator `snd`, extracting a sector-like boolean field as shown in Figure 3 (left). To do so, we use a special diffusion function `sum_or` working on `real,bool` pairs, which sums the first components, and apply disjunction to the second. In crowd steering, this pattern is useful to make people be aware of certain areas that the proposed path would cross, so as to support proper choices among alternatives [37]. Figure 12 (left) shows a pictorial representation.

However, our self-stabilisation result reveals a subtlety. Function `sum_or` has to be tuned by composing it with `pt_POSINF_TRUE` (which propagates the top value from the first to the second component of the pair), leading to function `sd_sum_or` (which is a stabilising diffusion). This is needed to make sure that the top value `<POSINF,TRUE>` of pairs of type `<real,bool>` is used when distance reaches `POSINF`: this is required to achieve progressiveness, and hence self-stabilisation. Without it, in the whole area where distance is `POSINF` we would have a behaviour similar to that of Example 3.8: in particular, if `c` is `true` and `i` is `POSINF` everywhere, both states where all nodes have second component equal to `true` (state s_1) and where all nodes have second component equal or `false` (state s_2) would be stable, and an even temporaneous flip of `c` to `false` in some node would make the system inevitable move to s_2—a clear indication of non self-stabilisation.

Note that `sector` function can be easily changed to propagate values of any sort by changing the type of the second component of pairs, and generalising over the `or` function. E.g., one could easily define a spreading of "set of values" representing the obstacles encountered during the spread.

$$\text{[T-PAIR]}\ \frac{\mathscr{T} \vdash \mathtt{e}_1 : \mathtt{T}_1 \quad \mathscr{T} \vdash \mathtt{e}_2 : \mathtt{T}_2}{\mathscr{T} \vdash \mathtt{<e_1,e_2>} : \mathtt{<T_1,T_2>}} \qquad \text{[T-FST]}\ \frac{\mathscr{T} \vdash \mathtt{e} : \mathtt{<T_1,T_2>}}{\mathscr{T} \vdash \mathtt{fst}\ \mathtt{e} : \mathtt{T}_1} \qquad \text{[T-SND]}\ \frac{\mathscr{T} \vdash \mathtt{e} : \mathtt{<T_1,T_2>}}{\mathscr{T} \vdash \mathtt{snd}\ \mathtt{e} : \mathtt{T}_2}$$

Figure 13: Type-checking rules for pair construction and deconstruction expressions (cf. Fig. 4)

Another interesting function working on pairs is `gradcast`, that is useful to let a gradient carry some information existing in the source. Expression `gradcast(i,j)`, where `i` is a 0-field as usual, results in a field of pairs in which the first component is the minimum distance from a source and the second component is the value of `j` at that source. Assuming `j` is a field of unique values in each node, e.g. a unique identifier ID, `gradcast` performs a partitioning behaviour (namely, a so-called Voronoi partition): its second component forms a partition of the network into regions based on "closeness" to sources, and each region of the partition holds the value of `j` in that source; first component still forms a gradient towards such sources. See Figure 12 (center). Again, function `add_to_1st` has to be changed to `sd_add_to_1st` to preserve self-stabilisation.

The remaining functions `dist`, `path` and `channel` are used to obtain a spatial pattern more heavily relying on multi-level composition, known as *channel* [46, 36]. Assume `i` and `j` are 0-fields, and suppose to be willing to steer people in complex and large environments from area `i` to destination `j`, i.e., from a node where `i` holds 0 to a node where `j` holds 0. Typically, it is important to activate the steering service (spreading information, providing signs, and detecting contextual information such as congestion) only along the shortest path, possibly properly extended of a distance width `w` to tolerate some randomness of people movement—see Figure 12 (right). Function `dist` uses `gradcast` to broadcasts the distance d between `i` and `j`—i.e., the minimum distance between a node where `i` holds 0 and a node where `j` holds 0. This is done by sending a gradcast from the source of `j` holding the value of `grad(i)` there, which is exactly the distance d. Function `path` simply marks as positive those nodes whose distance from the shortest path between `i` and `j` is smaller than `w`. Finally, function `channel` generates from `j` a gradient confined inside `path(i,j,w)`, which can be used to steer people towards the POI at `j` without escaping the path area.

6.3. **Type checking.** The typing rules for pair construction and deconstruction expressions are given in Figure 13. Note that adding these rules to the rules in Fig. 4 preserves the property that no choice may be done when building a derivation for a given type-checking judgment, so the type-checking rules straightforwardly describe a type-checking algorithm (cf. end of Section 3.1).

Example 6.1. Consider the library in Figure 11. The following predicates hold:
- *diffusion*(`sum_or`),
- *diffusion*(`sd_sum_or`),
- *diffusion*(`add_to_1st`), and
- *diffusion*(`sd_add_to_1st`)

Example 6.2. The library in Figure 11 type checks by using the ground types, sensors, and type-signatures for built-in functions in Figure 5.

6.4. **Device Computation.** The big-step operational semantics rules for pair construction and deconstruction expressions are given in Figure 14. Note that they are syntax directed (in particular, the first premise of rule [E-PAIR] ensures that there is no conflict with rule [E-VAL] of Fig. 6).

$$\texttt{v} ::= \texttt{g} \mid \texttt{<v,v>} \qquad \text{value}$$

$$\text{[E-PAIR]}\ \frac{\texttt{<e}_1,\texttt{e}_2\texttt{>}\ \text{not a value} \quad \sigma;\pi_1(\overline{\theta}) \vdash \texttt{e}_1 \Downarrow \eta_1 \quad \sigma;\pi_2(\overline{\theta}) \vdash \texttt{e}_2 \Downarrow \eta_2 \quad \texttt{v}_1 = \rho(\eta_1) \quad \texttt{v}_2 = \rho(\eta_2)}{\sigma;\overline{\theta} \vdash \texttt{<e}_1,\texttt{e}_2\texttt{>} \Downarrow \texttt{<v}_1,\texttt{v}_2\texttt{>}(\eta_1,\eta_2)}$$

$$\text{[E-FST]}\ \frac{\sigma;\pi_1(\overline{\theta}) \vdash \texttt{e} \Downarrow \eta \quad \texttt{<v}_1,\texttt{v}_2\texttt{>} = \rho(\eta)}{\sigma;\overline{\theta} \vdash \texttt{fst e} \Downarrow \texttt{v}_1(\eta)} \qquad \text{[E-SND]}\ \frac{\sigma;\pi_1(\overline{\theta}) \vdash \texttt{e} \Downarrow \eta \quad \texttt{<v}_1,\texttt{v}_2\texttt{>} = \rho(\eta)}{\sigma;\overline{\theta} \vdash \texttt{snd e} \Downarrow \texttt{v}_2(\eta)}$$

Figure 14: Big-step operational semantics for pair construction and deconstruction expressions (cf. Fig. 6)

6.5. **Sorts.** Each sort has the same structure of the type it refines. For instance, considering the sorts for ground types given in Figure 8, we can build $36(=6^2)$ *pair sorts* for the pair type `<real,real>`:

$$\begin{array}{llllll} \texttt{<nr,nr>}, & \texttt{<nr,znr>}, & \texttt{<nr,zr>}, & \texttt{<nr,zpr>}, & \texttt{<nr,pr>}, & \texttt{<nr,real>} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \texttt{<real,nr>}, & \texttt{<real,znr>}, & \texttt{<real,zr>}, & \texttt{<real,zpr>}, & \texttt{<real,pr>}, & \texttt{<real,real>} \end{array}$$

and $108(36*3)$ sorts for the type `<<real,real>,bool>`:

$$\texttt{<<nr,nr>,false>}, \quad \texttt{<<nr,znr>,false>}, \quad ..., \quad \texttt{<<real,real>,bool>}.$$

Subsorting between ground sorts can be lifted to the sorts for non-ground types by suitable subsorting rules. The following subsorting rule:

$$\frac{\text{[I-PAIR]} \quad \texttt{S}_1 \leq \texttt{S}'_1 \quad \texttt{S}_2 \leq \texttt{S}'_2}{\texttt{<S}_1,\texttt{S}_2\texttt{>} \leq \texttt{<S}'_1,\texttt{S}'_2\texttt{>}}$$

lifts subsorting between ground sorts to pair sorts by modelling pointwise ordering on pairs. Note that the subsorting relation is determined by the subsorting for ground sorts. Using the inclusions $\texttt{nr} \leq \texttt{znr}$ and $\texttt{true} \leq \texttt{bool}$, and the above rule it is possible to derive, e.g., the inclusion $\texttt{<<nr,nr>,true>} \leq \texttt{<<nr,znr>,bool>}$. Note that no choice may be done when building a derivation for a given subsorting judgment $\texttt{S}_1 \leq \texttt{S}_2$, so the subsorting rules (i.e, the rule [I-PAIR] and the subsorting for ground sorts) describe a deterministic algorithm.

Similarly, progressive subsorting between ground sorts is lifted to pair-sorts by the following progressive subsorting rule:

$$\frac{\text{[P-I-PAIR]} \quad \texttt{S}_1 \leq^{\text{progressive}} \texttt{S}'_1 \quad \texttt{S}_1 \leq^{\text{progressive}} \texttt{S}'_1}{\texttt{<S}_1,\texttt{S}_2\texttt{>} \leq^{\text{progressive}} \texttt{<S}'_1,\texttt{S}'_2\texttt{>}}$$

which (together with the progressive subsorting for ground sorts) describes a deterministic algorithm.

6.6. **Stabilising Diffusion predicate and Properties.** The stabilising diffusion predicate (Definition 4.2) and the stabilising diffusion condition (Section 4.3) are parametric in the set of types modeled by the calculus.

Example 6.3. The library in Figure 11 satisfies the stabilising diffusion condition. In particular, the following predicates hold:

- $\textit{stabilising}(\texttt{sd_sum_or},\texttt{<real,bool>}(\texttt{<real,bool>},\texttt{<pr,bool>}))$, and
- $\textit{stabilising}(\texttt{sd_add_to_1st},\texttt{<real,real>}(\texttt{<real,real>},\texttt{pr}))$.

The statements of device computation type preservation (Theorem 5.1), device computation termination (Theorem 5.2), and network self-stabilisation for programs with valid sort and stabilising assumptions (Theorem 5.3) are parametric in the set of types modeled by the calculus. The proofs (see Appendix A and Appendix B) are indeed given for the calculus with pairs—the cases for pair construction and deconstruction are straightforward by induction. The rest of the paper considers the calculus with pairs—in particular, the rules for checking sort and stabilising assumptions for diffusions are able to check the examples presented in Section 6.2.

7. On Checking the Stabilising-Diffusion Condition

The rest of the paper is devoted to illustrate a type-based analysis for checking the stabilising-diffusion condition (cf. Section 4.3) for the extension of the calculus with pairs introduced in Section 6.

7.1. **A Type-based approach for checking the Stabilising-Diffusion Condition.**

Recall that the stabilising-diffusion condition consists of two parts:

- validity of the sort-signature assumptions (Condition (1) of Section 4.3; and
- validity of the stabilising assumptions (Condition (2) of Section 4.3).

In order to check the stabilising-diffusion condition we assume that each program **P** comes with:

- a non-empty set of sort-signature assumptions $\textit{s-sigs}(\texttt{f})$ for each function $\texttt{f}$; and
- a possibly empty set of stabilising sort-signature assumptions $\textit{stb-s-sigs}(\texttt{f})$ for each diffusion $\texttt{f}$.

The assumptions $\textit{s-sigs}(\texttt{b})$ and $\textit{stb-s-sigs}(\texttt{b})$ for the built-in functions $\texttt{b}$ are considered valid—they should come with the definition of the language. Instead, the validity of the assumptions $\textit{s-sigs}(\texttt{d})$ and $\textit{stb-s-sigs}(\texttt{d})$ for the user-defined functions $\texttt{d}$ must be checked—these assumptions could be either (possibly partially) provided by the user or automatically inferred.[6]

7.1.1. *On checking Condition (1) of Section 4.3.*

Section 8 introduces a sort-checking system that checks Condition (1) of Section 4.3. Namely, given a program (or library) **P**, it checks that: (*i*) each user-defined function $\texttt{d}$ in **P** has all the sort signatures in $\textit{s-sigs}(\texttt{d})$ and, if $\texttt{d}$ is a diffusion, it has also the sort signatures in $\textit{stb-s-sigs}(\texttt{d})$; and (*ii*) every diffusion-expressions $\{\texttt{e}_1 : \texttt{f}(@, \texttt{e}_2, ..., \texttt{e}_n)\}$ occurring in **P** is sort-checked by considering for $\texttt{f}$ only the sort-signatures in $\textit{stb-s-sigs}(\texttt{f})$. The soundness of the sort-checking system (shown in Section 8.6) guarantees that, if the check is passed, then for every diffusion-expressions $\{\texttt{e}_1 : \texttt{f}(@, \texttt{e}_2, ..., \texttt{e}_n)\}$ occurring in the **P** there is a sort-signature $\texttt{S}(\texttt{S}_1, \ldots, \texttt{S}_n) \in \textit{stb-s-sigs}(\texttt{f})$ such that the evaluation of the subexpression $\texttt{e}_i$ yields a value $\texttt{v}_i \in [\![\texttt{S}_i]\!]$ $(1 \leq i \leq n)$. I.e., Condition (1) of Section 4.3 holds.

---

[6]The naive inference approach, that is: inferring $\textit{s-sigs}(\texttt{d})$ by checking all the possible refinements of the type-signature of $\texttt{d}$ is linear in the number of elements of **sort-signatures**$(\textit{t-sig}(\texttt{d}))$. Some optimizations are possible. We do not address this issue in the paper.

7.1.2. *On checking Condition (2) of Section 4.3.* Note that, if Condition (1) of Section 4.3 has been checked by the sort-checking system of Section 8, then in order to check Condition (2) of Section 4.3 holds it is enough to check that for each user-defined diffusion $\mathtt{d}$, each sort-signature $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n) \in \mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})$ is stabilising for $\mathtt{d}$.

In order to check that for each user-defined diffusion $\mathtt{d}$ of signature $\mathtt{T}_1(\mathtt{T}_1,\ldots,\mathtt{T}_n)$ each sort-signature $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n) \in \mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})$ is stabilising for $\mathtt{d}$, we introduce additional requirements. Namely, we require that for every user-defined diffusion $\mathtt{d}$ such that $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{d}) \neq \emptyset$:

(1) there exists a sort-signature $\mathtt{S}(\cdots) \in \mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})$ such that $\mathtt{S}'(\cdots) \in \mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})$ implies $\mathtt{S} \leq^{\text{progressive}} \mathtt{S}'$; and
(2) if $\mathtt{T}_1$ is not ground (i.e., if it is a pair type), then the user-defined diffusion $\mathtt{d}$ is of the form

$$\mathtt{def}\ \mathtt{T}_1\ \mathtt{d}(\mathtt{T}_1\ \mathtt{x}_1,\ldots,\mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{is}\ \mathtt{pt}[\top_{\mathtt{S}}](\mathtt{f}(\mathtt{x}_1,\ldots,\mathtt{x}_n)) \tag{7.1}$$

where
- $\mathtt{pt}[\top_{\mathtt{S}}]$ (defined in Section 7.2) is a pure function of sort-signature $\mathtt{S}(\mathtt{S})$,
- $\mathtt{f}$ is a diffusion, and
- if $\mathtt{f}$ is user-defined then $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{f}) = \emptyset$.

Note that the above additional requirements can be checked automatically.

In the rest of this section we first (in Section 7.2) introduce some auxiliary definitions (including, for each sort $\mathtt{S}$, the definition of the pure function $\mathtt{pt}[\top_{\mathtt{S}}]$); then (in Section 7.3) we introduce the notion of *!-prestabilising diffusion with respect to a progressive sort-signature* $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n)$ and show that

- $\mathtt{S}$ ground implies that: if $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n)$ is $\mathtt{!}$-prestabilising for the user-defined diffusion $\mathtt{d}$ then $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n)$ is stabilising for $\mathtt{d}$;
- if $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n)$ is $\mathtt{!}$-prestabilising for the diffusion $\mathtt{f}$ then $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n)$ is stabilising for the user-defined diffusion $\mathtt{d}$ displayed in Equation 7.1; and

finally (in Section 7.4) we introduce *annotated sort-signatures* and *annotated sorts* as convenient notations to be used in writing type-based rules for checking $\mathtt{!}$-prestabilisation.

Section 9 introduces an annotated sort checking system that checks that for each user-defined diffusion $\mathtt{d}$ of signature $\mathtt{T}_1(\mathtt{T}_1,\ldots,\mathtt{T}_n)$ and for each sort-signature $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n) \in \mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})$:

- $\mathtt{S}$ ground implies that $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n)$ is $\mathtt{!}$-prestabilising for $\mathtt{d}$; and
- $\mathtt{S}$ not ground implies that $\mathtt{S}(\mathtt{S}_1,\ldots,\mathtt{S}_n)$ is $\mathtt{!}$-prestabilising for the diffusion $\mathtt{f}$ occurring in Equation 7.1.

The soundness of the annotated sort checking system (shown in Section 9.5) guarantees that, if the check is passed, then Condition (2) of Section 4.3 holds.

7.2. **Auxiliary definitions.** For any type $\mathtt{T}$ the *leftmost-as-key preorder* $\leq^1_{\mathtt{T}}$ is the preorder that weakens the order $\leq_{\mathtt{T}}$ by considering each pair as a record where the leftmost ground element is the key. It is defined by:

- $\mathtt{v} \leq^1_{\mathtt{T}} \mathtt{v}'$ if $\mathtt{v} \leq_{\mathtt{T}} \mathtt{v}'$, where $\mathtt{T}$ is a ground type; and
- $\mathtt{<v_1,v_2>} \leq^1_{\mathtt{<T_1,T_2>}} \mathtt{<v'_1,v'_2>}$ if $\mathtt{v}_1 \leq^1_{\mathtt{T}_1} \mathtt{v}'_1$.

Note that that the leftmost-as-key preorder is total, i.e., for every $\mathtt{v},\mathtt{v}' \in [\![\mathtt{T}]\!]$ we have that either $\mathtt{v} \leq^1_{\mathtt{T}} \mathtt{v}'$ or $\mathtt{v}' \leq^1_{\mathtt{T}} \mathtt{v}$ holds. We write $\mathtt{v} =^1_{\mathtt{T}} \mathtt{v}'$ to mean that both $\mathtt{v} \leq^1_{\mathtt{T}} \mathtt{v}'$ and $\mathtt{v}' \leq^1_{\mathtt{T}} \mathtt{v}$ hold. Of course $\mathtt{v} =^1_{\mathtt{T}} \mathtt{v}'$ does not imply $\mathtt{v} =_{\mathtt{T}} \mathtt{v}'$. Note also that $\mathtt{v} <^1_{\mathtt{T}} \mathtt{v}'$ implies $\mathtt{v} <_{\mathtt{T}} \mathtt{v}'$.

For every sort $\mathtt{S}$ of $\mathtt{T}$, we wrote $\leq^1_{\mathtt{S}}$ to denote the restriction of $\leq^1_{\mathtt{T}}$ to $[\![\mathtt{S}]\!]$. According to the previous definition, we define the *key* of a sort $\mathtt{S}$ as the the leftmost ground sort occurring in $\mathtt{S}$,

and the *key* of a value $\mathtt{v}$ as the the leftmost ground value occurring in $\mathtt{v}$. The *key* mappings can be inductively defined as follows:

- $\mathit{key}(\mathtt{S}) = \mathtt{S}$, if $\mathtt{S}$ is a ground sort
- $\mathit{key}(\mathtt{S}) = \mathit{key}(\mathtt{S}_1)$, if $\mathtt{S} = \mathtt{<}\mathtt{S}_1, \cdots\mathtt{>}$

and

- $\mathit{key}(\mathtt{v}) = \mathtt{v}$, if $\mathtt{v}$ is a ground value
- $\mathit{key}(\mathtt{v}) = \mathit{key}(\mathtt{v}_1)$, if $\mathtt{v} = \mathtt{<}\mathtt{v}_1, \cdots\mathtt{>}$.

Note that, for every $\mathtt{v}$ and $\mathtt{v}'$ of sort $\mathtt{S}$ it holds that:

$$\mathtt{v} \leq^1_{\mathtt{S}} \mathtt{v}' \qquad \text{if and only if} \qquad \mathit{key}(\mathtt{v}) \leq_{\mathit{key}(\mathtt{S})} \mathit{key}(\mathtt{v}'). \tag{7.2}$$

For every sort $\mathtt{S}$ of type $\mathtt{T}$ we write $\mathtt{pt}[\top_{\mathtt{S}}]$ to denote the pure function (which satisfies the sort-signature $\mathtt{S}(\mathtt{S})$), that maps the elements $\mathtt{v}$ such that $\mathtt{v} =^1_{\mathtt{S}} \top_{\mathtt{S}}$ to $\top_{\mathtt{S}}$ (i.e., that propagates the top value from the key of a pair value to all the other components of the value), and is the identity otherwise. Note that the function $\mathtt{pt}[\top_{\mathtt{S}}]$ can be inductively defined as: $\mathtt{def\ T\ pt}[\top_{\mathtt{S}}](\mathtt{T\ x})\ \mathtt{is\ e}$, where

$$\mathtt{e} = \begin{cases} \mathtt{x} & \text{if } \mathtt{S} \text{ is ground} \\ \mathtt{pt}[\top_{\mathtt{S}'}](\mathtt{fst\ x}) = \top_{\mathtt{S}'})\ ?\ \top_{\mathtt{S}}\ :\mathtt{x} & \text{if } \top_{\mathtt{S}} = \mathtt{<}\top_{\mathtt{S}'}, \cdots\mathtt{>} \end{cases}$$

For every diffusion $\mathtt{f}$ with signature$\mathtt{T}_1(\mathtt{T}_1, \ldots, \mathtt{T}_n)$ and sort signature $\mathtt{S}(\mathtt{S}_1, \ldots, \mathtt{S}_n)$ $(n \geq 1)$ we write $\mathtt{sd}[\mathtt{f}, \top_{\mathtt{S}}]$ to denote the composition of $\mathtt{f}$ and $\mathtt{pt}[\top_{\mathtt{S}}]$, which is the diffusion (which satisfies the sort-signature $\mathtt{S}(\mathtt{S}_1, \ldots, \mathtt{S}_n)$) defined as follows:

$$\mathtt{def\ T_1\ sd}[\mathtt{f}, \top_{\mathtt{S}}](\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{is\ pt}[\top_{\mathtt{S}}](\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n))$$

7.3. **`!`-Prestabilising Diffusions and `?`-Prestabilising Diffusions.** A π-prestabilising diffusion is a diffusion whose progressiveness behaviour is expressed by the annotation π that ranges over `!` (for *certainly prestabilising*) and `?` (for *possibly prestabilising*), as illustrated by the following definition.

Definition 7.1 (π-prestabilising diffusion)**.** A diffusion $\mathtt{f}$ is *π-prestabilising with respect to the progressive sort signature* $\mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}}) \in$ **sort-signatures**$(\mathtt{f})$ (notation *π-prestabilising*$(\mathtt{f}, \mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}}))$) if for any $\overline{\mathtt{v}} \in [\![\overline{\mathtt{S}}]\!]$:

(1) if $\pi = \mathtt{!}$ then
  - $\mathtt{v} <^1_{\mathtt{S}_1} \mathtt{v}'$ and $[\![\mathtt{f}]\!](\mathtt{v}, \overline{\mathtt{v}}) = \mathtt{v}'' \neq^1_{\mathtt{S}_1} \top_{\mathtt{S}_1}$ imply $\mathtt{v}'' <^1_{\mathtt{S}_1} [\![\mathtt{f}]\!](\mathtt{v}', \overline{\mathtt{v}})$;
  - for all $\mathtt{v} \in [\![\mathtt{S}_1]\!] - \{\top_{\mathtt{S}_1}\}$, $\mathtt{v} <^1_{\mathtt{S}_1} [\![\mathtt{f}]\!](\mathtt{v}, \overline{\mathtt{v}})$;

(2) if $\pi \in \{\mathtt{!}, \mathtt{?}\}$ then
  - $\mathtt{v} \leq^1_{\mathtt{S}_1} \mathtt{v}'$ implies $[\![\mathtt{f}]\!](\mathtt{v}, \overline{\mathtt{v}}) \leq^1_{\mathtt{S}_1} [\![\mathtt{f}]\!](\mathtt{v}', \overline{\mathtt{v}})$;
  - for all $\mathtt{v} \in [\![\mathtt{S}_1]\!]$, $\mathtt{v} \leq^1_{\mathtt{S}_1} [\![\mathtt{f}]\!](\mathtt{v}, \overline{\mathtt{v}})$.

We say that the sort-signature $\mathtt{S}(\overline{\mathtt{S}})$ is *π-prestabilising for* $\mathtt{f}$ to mean that *π-prestabilising*$(\mathtt{f}, \mathtt{S}(\overline{\mathtt{S}}))$ holds, and write π-**prestabilising-sort-signatures**$(\mathtt{f})$ to denote set of the π-prestabilising sort-signatures for $\mathtt{f}$.

Recall the definition of $\mathtt{sd}[\mathtt{f}, \top_{\mathtt{S}}]$ given at the end of Section 7.2. The following proposition guarantees that if $\mathtt{S}(\overline{\mathtt{S}})$ is `!`-prestabilising for the diffusion $\mathtt{f}$ then:

- $\mathtt{S}$ ground implies that $\mathtt{S}(\overline{\mathtt{S}})$ is stabilising for $\mathtt{f}$; and
- $\mathtt{S}(\overline{\mathtt{S}})$ is stabilising for the user-defined diffusion $\mathtt{d}$ displayed in Equation 7.1 of Section 7.2.

Proposition 7.2. (1) *if S is ground then: $!$-stabilising$(f, S(\overline{S}))$ implies stabilising$(f, S(\overline{S}))$, i.e,*

$$\textit{!-}\textbf{\textit{prestabilising-sort-signatures}}(f) \subseteq \textbf{\textit{stabilising-sort-signatures}}(f).$$

(2) *!-stabilising*$(f, S(\overline{S}))$ *implies stabilising*$(\mathit{sd}[f, \top_S], S(\overline{S}))$*, i.e,*

$$\textit{!-}\textbf{\textit{prestabilising-sort-signatures}}(f) \subseteq \textbf{\textit{stabilising-sort-signatures}}(\mathit{sd}[f,S]).$$

Proof. Straightforward from Definition 7.1, Definition 4.2, and the definition of $\mathtt{sd}[\mathtt{f},\mathtt{S}]$. □

Example 7.3. Consider the libraries in Figure 2 and 11. The following predicates hold (cf. Examples 4.3 and 6.3):

- `!`-*stabilising*(+, `real(real,pr)`) and `?`-*stabilising*(+, `real(pr,zpr)`)
- `!`-*stabilising*(`max`, `pr(znr,pr)`) and `?`-*stabilising*(`max`, `real(real,real)`), where `max` is the the binary maximum function:

```
def max(real x, real y) is x < y ?  y :  x
```

- `?`-*stabilising*(`id`, `real(real)`)
- `?`-*stabilising*(`restrict`, `real(real,bool)`)
- `!`-*stabilising*(`restrictSum`, `real(real,pr,bool)`)
- `!`-*stabilising*(`sum_or`, `<real,bool>(<real,bool>,<pr,bool>)`)
- `!`-*stabilising*(`add_to_1st`, `<real,real>(<real,real>,pr)`)

7.4. **Annotated Sort-Signatures and Annotated Sorts.** In order to be able to write type-based rules for checking π-stabilisation we introduce, as convenient notations, *annotated sort-signatures* and *annotated sorts*.

An *annotated sort-signature* $\mathtt{S}(\overline{\mathtt{S}})[\pi]$ is a progressive sort-signature (cf. Definition 4.4) with a π annotation. It provides a convenient notation to express the fact that the predicate π-*prestabilising*$(\mathtt{f},\mathtt{S}(\overline{\mathtt{S}}))$ holds. Namely, we say that a diffusion `f` *has* (or *satisfies*) the annotated sort-signature $\mathtt{S}(\overline{\mathtt{S}})[\pi]$ to mean that the predicate π-*prestabilising*$(\mathtt{f},\mathtt{S}(\overline{\mathtt{S}}))$ holds. We write π-**annotated-sort-signatures**(`f`) to denote the set of the annotated sort-signatures with annotation π that are satisfied by the diffusion `f`, and we write **annotated-sort-signatures**(`f`) to denote `!`-**annotated-sort-signatures**(`f`) ∪ `?`-**annotated-sort-signatures**(`f`).

The *support* of an annotated sort signature $\mathtt{S}(\overline{\mathtt{S}})[\pi]$ is the progressive sort signature $\mathtt{S}(\overline{\mathtt{S}})$. Given an annotated sort-signature $\mathtt{S}(\overline{\mathtt{S}})[\pi]$ we write $|\mathtt{S}(\overline{\mathtt{S}})[\pi]|$ to denote its support. Note that, according to the above definitions, the mapping π-**annotated-sort-signatures**(·) provides the same information of the mapping π-**prestabilising-sort-signatures**(·) introduced in Section 7.3, i.e., π-**prestabilising-sort-signatures**(`f`) = |π-**annotated-sort-signatures**(`f`)|.

Given a diffusion type-signature $\mathtt{T}(\overline{\mathtt{T}})$ (cf. Definition 3.2) we write **annotated-sort-signatures**$(\mathtt{T}(\overline{\mathtt{T}}))$ to denote the (set of) annotated sort-signatures that refine it, i.e., the set

$$\{\mathtt{S}(\mathtt{S}'\overline{\mathtt{S}})[\pi] \mid \mathtt{S}(\mathtt{S}'\overline{\mathtt{S}}) \in \textbf{progressive-sort-signatures}(\mathtt{T}(\overline{\mathtt{T}})) \text{ and } \pi \in \{\mathtt{!},\mathtt{?}\}\}.$$

Recall the stabilising subsigning partial order between progressive signatures introduced at the end of Section 4.2. The following order between progressiveness annotations, that we call *subannotating* relation and denote by ≤,

```
?
↑
!
```

induces the following partial order between annotated sort-signatures, that we call *annotated subsigning*:

$$\text{[I-A-SIG]}\ \frac{\mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}}) \leq^{\text{stabilising}} \mathtt{S}'(\mathtt{S}'_1\overline{\mathtt{S}}') \quad \pi \leq \pi'}{\mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}})\,[\pi] \leq \mathtt{S}'(\mathtt{S}'_1\overline{\mathtt{S}}')\,[\pi']}$$

The following proposition shows that annotated subsigning captures the natural implication relation between π-prestabilisation properties.

Proposition 7.4 (Sounness of annotated subsigning)**.** *If the diffusion f satisfies the the annotated sort-signature $\mathtt{S}(\overline{\mathtt{S}})\,[\pi]$ and $\mathtt{S}(\overline{\mathtt{S}})\,[\pi] \leq \mathtt{S}'(\overline{\mathtt{S}}')\,[\pi']$, then f satisfies $\mathtt{S}'(\overline{\mathtt{S}}')\,[\pi']$.*

Proof. Straightforward from Definition 7.1 and the definition of annotated subsigning (Rule [I-A-SIG] above). □

We say that an value $\mathtt{v}$ has (or satisfies) *annotated sort* $\mathtt{S}'_1\,[\pi]$ to mean that $\mathtt{v}$ has sort $\mathtt{S}'_1$, and

- the application of any diffusion with annotated sort signature $\mathtt{S}(\mathtt{S}_1,...,\mathtt{S}_n)\,[\pi]$ such that both $key(\mathtt{S}'_1) \leq^{\text{progressive}} key(\mathtt{S}_1)$ and $\mathtt{S}'_1 \leq \mathtt{S}_1$ hold
- to $\mathtt{v}$ and to values $\mathtt{v}_2,\ldots,\mathtt{v}_n$ of sorts $\mathtt{S}'_2,...,\mathtt{S}'_n$ (respectively) such that $\mathtt{S}'_2 \leq \mathtt{S}_2,...,\mathtt{S}'_n \leq \mathtt{S}_n$,

produces a result of annotated sort $\mathtt{S}\,[\pi'']$, where $\pi'' = \pi(\pi')$. According to this definition, the following property holds.

Proposition 7.5 (Annotated sorts for ground values)**.** *For every sort $\mathtt{S} \in \textbf{sorts}(\mathtt{T})$ the maximum element $\mathtt{v}$ of $[\![\mathtt{S}]\!]$ w.r.t. $\leq_{\mathtt{S}}$ has both sort $\mathtt{S}\,[!]$ and sort $\mathtt{S}\,[?]$.*

Proof. Straightforward from Definition 7.1. □

The following partial order between annotated sorts, that we call *annotated subsorting*, models the natural implication between the properties they represent:

$$\text{[I-A-SORT]}\ \frac{key(\mathtt{S}) \leq^{\text{progressive}} key(\mathtt{S}') \quad \mathtt{S} \leq \mathtt{S}' \quad \pi \leq \pi'}{\mathtt{S}\,[\pi] \leq \mathtt{S}'\,[\pi']}$$

The *support* of an annotated sort $\mathtt{S}\,[\pi]$ is the sort $\mathtt{S}$. Given an annotated sort $\mathtt{A}$ we write $|\mathtt{A}|$ to denote its support.

8. Checking Sort-Signature Assumptions for User-Defined Functions

In this section we present a decidable sort-checking system that guarantees that if a program (or library) **P** is *well-sorted* (i.e., it can be successfully checked by the rules of the system) then Condition (1) of Section 4.3 holds. [7]

We first introduce some auxiliary definitions (in Section 8.1); then we consider the issue of associating sorts to values and sensors (in Section 8.2), sort-signatures to functions (in Section 8.3) and stabilising sort-signatures to diffusions (in Section 8.4); and finally we present a decidable sort system for checking the correctness of sort-signature declarations for user-defined functions (in Section 8.5) and show that it is sound (in Section 8.6).

[7] Note that, in this section, we do not use the additional requirements (1) and (2) introduced at the beginning of Section 7.1.2. This generality might become useful since Condition (2) of Section 4.3 might be checked by using a technique different from the one presented in Section 7.

8.1. **Auxiliary Definitions.** The auxiliary definitions presented in this section will be used (in Section 8.5) to formulate the sort-checking rules for function applications and spreading expressions.

Recall that for every type $\mathtt{T}$ and type-signature $\mathtt{T}(\overline{\mathtt{T}})$ both $(\mathbf{sorts}(\mathtt{T}), \leq)$ and $(\mathbf{sort\text{-}signatures}(\mathtt{T}(\overline{\mathtt{T}})), \leq)$ are partial orders (cf. Section 4.1). Sort checking an expression $\mathtt{e}$ of type $\mathtt{T}$ amounts to compute an *abstract interpretation* [15] over these partial orders.

Given a partial order $(P, \leq)$ and a subset Q of P we say that:

- an element $q_0 \in Q$ is *minimal in* Q to mean that: if $q \in Q$ and $q \leq q_0$ then $q = q_0$—the set of the minimal elements of Q is denoted by $minimals(Q)$;
- Q is *minimised* to mean that every $q \in Q$ is minimal in Q, i.e., that $Q = minimals(Q)$.

Given a set of sort-signatures $Q \subseteq \mathbf{sort\text{-}signatures}(\mathtt{T}(\overline{\mathtt{T}}))$ and some sorts $\overline{\mathtt{S}}' \in \mathbf{sort\text{-}signatures}(\overline{\mathtt{T}})$, consider the (possibly empty) subset of Q defined as follows:

$$Q(\overline{\mathtt{S}}') = \{\mathtt{S}(\overline{\mathtt{S}}) \in Q \mid \overline{\mathtt{S}}' \leq \overline{\mathtt{S}}\}.$$

We say that Q is *deterministic*, notation $deterministic(Q)$, to mean that for all sorts $\overline{\mathtt{S}}'$ there exists a sort-signature $\mathtt{S}(\overline{\mathtt{S}}) \in Q(\overline{\mathtt{S}}')$, called the *most specific sort-signature for* $\overline{\mathtt{S}}'$ *in* Q, such that:

$$\text{for all } \mathtt{S}''(\overline{\mathtt{S}}'') \in Q(\overline{\mathtt{S}}') \text{ it holds that } \mathtt{S} \leq \mathtt{S}''.$$

The mapping $ms(Q, \overline{\mathtt{S}}')$, given a deterministic set of sort-signatures $Q \subseteq \mathbf{sort\text{-}signatures}(\mathtt{T}(\overline{\mathtt{T}}))$ and some sorts $\overline{\mathtt{S}}' \in \mathbf{sort\text{-}signatures}(\overline{\mathtt{T}})$, returns the *most specific sort-signature for* $\overline{\mathtt{S}}'$ *in* Q if $Q(\overline{\mathtt{S}}')$ is not empty, and is undefined otherwise.

8.2. **Sorts for Values and Sensors.** We assume a mapping $sort(\cdot)$ that associates:

- to each ground value $\mathtt{g}$ the minimum (w.r.t. $\leq$) sort $sort(\mathtt{g})$ in $\mathbf{sorts}(\mathtt{g})$, and
- to each sensor s the minimum (w.r.t. $\leq$) sort $sort(s)$ in $\mathbf{sorts}(type(s))$ such that $\mathtt{g} \in [\![sort(s)]\!]$ for every ground value $\mathtt{g}$ that may be returned by s.

Example 8.1. Figure 15 illustrates the sorts for the ground values and sensors used in the examples introduced throughout the paper.

8.3. **Sort-signatures for Functions.** We assume a mapping $s\text{-}sigs(\cdot)$ that associates to each built-in function $\mathtt{b}$ a set of sort-signatures $s\text{-}sigs(\mathtt{b}) \subseteq \mathbf{sort\text{-}signatures}(\mathtt{b})$ such that the following conditions are satisfied:

- $s\text{-}sigs(\mathtt{b})$ is non-empty, minimised and deterministic, and
- $s\text{-}sigs(\mathtt{b})$ represents all the sort-signatures satisfied by $\mathtt{f}$, i.e., for each $\mathtt{S}'(\overline{\mathtt{S}}') \in \mathbf{sort\text{-}signatures}(\mathtt{b})$ there exists $\mathtt{S}(\overline{\mathtt{S}}) \in s\text{-}sigs(\mathtt{b})$ such that $\mathtt{S}(\overline{\mathtt{S}}) \leq \mathtt{S}'(\overline{\mathtt{S}}')$ holds.

Note that the first of the above two conditions on the mapping $s\text{-}sigs(\mathtt{b})$ can be checked automatically.

Example 8.2. Figure 16 illustrates the sort-signatures for built-in functions used in the examples introduced throughout the paper.

We also assume that the mapping $s\text{-}sigs(\cdot)$ associates to each user-defined function $\mathtt{d}$ a set of sort-signatures $s\text{-}sigs(\mathtt{d}) \subseteq \mathbf{sort\text{-}signatures}(t\text{-}sig(\mathtt{d}))$ such that the following conditions are satisfied:

- $s\text{-}sigs(\mathtt{d})$ is non-empty, minimised and deterministic, and
- $s\text{-}sigs(\mathtt{d})$ contains at least a sort-signature which is smaller than the type-signature $t\text{-}sig(\mathtt{d})$, i.e., there exists $\mathtt{S}(\overline{\mathtt{S}}) \in s\text{-}sigs(\mathtt{d})$ such that $\mathtt{S}(\overline{\mathtt{S}}) \leq t\text{-}sig(\mathtt{d})$ holds.

Note that the above two conditions on the mapping $s\text{-}sigs(\mathtt{d})$ can be checked automatically.

Ground values sort:

$$
\begin{array}{lcl}
\mathit{sort}(\mathtt{FALSE}) & = & \mathtt{false} \\
\mathit{sort}(\mathtt{TRUE}) & = & \mathtt{true} \\
\mathit{sort}(\mathtt{g}) & = & \mathtt{nr}, \quad \text{if } \mathit{type}(\mathtt{g}) = \mathtt{real} \text{ and } \mathtt{g} < 0 \\
\mathit{sort}(\mathtt{g}) & = & \mathtt{zr}, \quad \text{if } \mathtt{g} = 0 \\
\mathit{sort}(\mathtt{g}) & = & \mathtt{pr}, \quad \text{if } \mathit{type}(\mathtt{g}) = \mathtt{real} \text{ and } \mathtt{g} > 0
\end{array}
$$

Sensors sort:

$$
\begin{array}{lcl}
\mathit{sort}(\mathtt{\#src}) & = & \mathtt{zpr} \\
\mathit{sort}(\mathtt{\#dist}) & = & \mathtt{pr}
\end{array}
$$

Figure 15: Sorts for ground values and sensors used in the examples (cf. Figure 5)

8.4. **Stabilising Sort-signatures for Diffusions.** We assume a mapping $\mathit{stb\text{-}s\text{-}sigs}(\cdot)$ that associates to each built-in diffusion $\mathtt{b}$ a (possibly empty) set of sort-signatures $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{b})$ such that the following conditions are satisfied:

- $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{b})$ is minimised and deterministic;
- $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{b}) \subseteq$ **stabilising-sort-signatures**$(\mathtt{b})$; and
- $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{b})$ represents all the stabilising sort-signatures satisfied by $\mathtt{b}$, i.e., for each $\mathtt{S}'(\overline{\mathtt{S}}') \in$ **stabilising-sort-signatures**$(\mathtt{b})$ there exists $\mathtt{S}(\overline{\mathtt{S}}) \in \mathit{stb\text{-}s\text{-}sigs}(\mathtt{b})$ such that $\mathtt{S}(\overline{\mathtt{S}}) \leq \mathtt{S}'(\overline{\mathtt{S}}')$ holds.

Note that the first of the above three conditions on the mapping $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{b})$ can be checked automatically.

Example 8.3. Figure 17 gives the stabilising sort-signatures for the built-in diffusions $\mathtt{b}$ used in the examples introduced throughout the paper—the built-in diffusions without stabilising sort-signatures are omitted. Note that $\mathit{stb\text{-}s\text{-}sigs}(+) \not\subseteq \mathit{s\text{-}sigs}(+)$, since the stabilising sort-signature $\mathtt{real}(\mathtt{real},\mathtt{pr}) \in \mathit{stb\text{-}s\text{-}sigs}(+)$ is not minimal in $\mathit{s\text{-}sigs}(+) \cup \{\mathtt{real}(\mathtt{real},\mathtt{pr})\}$ and therefore it cannot be included in $\mathit{s\text{-}sigs}(+)$—it would break both the requirement that $\mathit{s\text{-}sigs}(+)$ must be minimised and deterministic (condition (1) at the beginning of Section 7.1.2).

We also assume that the mapping $\mathit{stb\text{-}s\text{-}sigs}(\cdot)$ associates to each user-defined diffusion $\mathtt{d}$ a (possibly empty) set of stabilising sort-signatures $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{d}) \subseteq$ **stabilising-sort-signatures**$(\mathtt{d})$ such that the following conditions are satisfied:

- $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})$ is minimised and deterministic, and
- $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})$ is implied by $\mathit{s\text{-}sigs}(\mathtt{d})$, i.e., for each $\mathtt{S}'(\overline{\mathtt{S}}') \in \mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})$ there exists $\mathtt{S}(\overline{\mathtt{S}}) \in \mathit{s\text{-}sigs}(\mathtt{d})$ such that $\mathtt{S}(\overline{\mathtt{S}}) \leq \mathtt{S}'(\overline{\mathtt{S}}')$.

Note that the above two conditions on the mapping $\mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})$ can be checked automatically.

Example 8.4. We assume that for the user-defined functions $\mathtt{d}$ used in the examples introduced throughout the paper

$$\mathit{s\text{-}sigs}(\mathtt{d}) = \mathit{minimals}(\{\mathit{t\text{-}sig}(\mathtt{d})\} \cup \mathit{stb\text{-}s\text{-}sigs}(\mathtt{d})).$$

Figure 18 gives minimised deterministic sets of stabilising sort-signatures that allow to successfully check the user-defined diffusions $\mathtt{d}$ used in the examples introduced in the paper—note that both the additional requirements (1) and (2) given at the beginning of Section 7 are satisfied.

$$
\begin{array}{lcl}
\textit{s-sigs}(\texttt{not}) & = & \texttt{true}(\texttt{false}),\\
& & \texttt{false}(\texttt{true}),\\
& & \texttt{bool}(\texttt{bool})\\
\textit{s-sigs}(\texttt{or}) & = & \texttt{false}(\texttt{false},\texttt{false}),\\
& & \texttt{true}(\texttt{true},\texttt{bool}),\\
& & \texttt{true}(\texttt{bool},\texttt{true}),\\
& & \texttt{bool}(\texttt{bool},\texttt{bool})\\
\textit{s-sigs}(-) & = & \texttt{nr}(\texttt{pr}),\\
& & \texttt{znr}(\texttt{zpr}),\\
& & \texttt{zr}(\texttt{zr}),\\
& & \texttt{zpr}(\texttt{znr}),\\
& & \texttt{pr}(\texttt{nr}),\\
& & \texttt{real}(\texttt{real})\\
\textit{s-sigs}(+) & = & \texttt{nr}(\texttt{nr},\texttt{znr}),\\
& & \texttt{nr}(\texttt{znr},\texttt{nr}),\\
& & \texttt{znr}(\texttt{znr},\texttt{znr}),\\
& & \texttt{zr}(\texttt{zr},\texttt{zr}),\\
& & \texttt{zpr}(\texttt{zpr},\texttt{zpr}),\\
& & \texttt{pr}(\texttt{zpr},\texttt{pr}),\\
& & \texttt{pr}(\texttt{pr},\texttt{zpr}),\\
& & \texttt{real}(\texttt{real},\texttt{real})\\
\textit{s-sigs}(=) & = & \texttt{false}(\texttt{znr},\texttt{pr}),\\
& & \texttt{false}(\texttt{nr},\texttt{zpr}),\\
& & \texttt{false}(\texttt{zpr},\texttt{nr}),\\
& & \texttt{false}(\texttt{pr},\texttt{znr}),\\
& & \texttt{true}(\texttt{zr},\texttt{zr}),\\
& & \texttt{bool}(\texttt{real},\texttt{real})\\
\textit{s-sigs}(<) & = & \texttt{false}(\texttt{zpr},\texttt{nr}),\\
& & \texttt{false}(\texttt{pr},\texttt{znr}),\\
& & \texttt{false}(\texttt{zr},\texttt{zr}),\\
& & \texttt{true}(\texttt{nr},\texttt{zpr}),\\
& & \texttt{true}(\texttt{znr},\texttt{pr}),\\
& & \texttt{bool}(\texttt{real},\texttt{real})
\end{array}
$$

Figure 16: Sort-signatures for built-in functions used in the examples (cf. Figure 5)

$$
\begin{array}{lcll}
\textit{stb-s-sigs}(\texttt{or}) & = & \texttt{false}(\texttt{false},\texttt{false}), & \\
& & \texttt{true}(\texttt{true},\texttt{bool}), & \\
& & \texttt{true}(\texttt{bool},\texttt{true}) & \\
\textit{stb-s-sigs}(+) & = & \texttt{zr}(\texttt{zr},\texttt{zr}), & \\
& & \texttt{pr}(\texttt{zpr},\texttt{pr}), & \\
& & \texttt{real}(\texttt{real},\texttt{pr}) & (\notin \textit{s-sigs}(+))
\end{array}
$$

Figure 17: Stabilising sort-signatures for built-in functions used in the examples (cf. Figure 16)

$$
\begin{array}{lcl}
\textit{stb-s-sigs}(\texttt{restrictSum}) & = & \texttt{real}(\texttt{real},\texttt{pr},\texttt{bool}),\\
\textit{stb-s-sigs}(\texttt{sd_sum_or}) & = & \texttt{<real,bool>}(\texttt{<real,bool>},\texttt{<pr,bool>})\\
\textit{stb-s-sigs}(\texttt{sd_add_to_1st}) & = & \texttt{<real,real>}(\texttt{<real,real>},\texttt{pr})
\end{array}
$$

Figure 18: Stabilising sort-signatures for the user-defined functions used in the examples

Expression sort checking: $\boxed{\mathscr{S} \vdash \mathtt{e} : \mathtt{S}}$

$$
\frac{\text{[S-VAR]}}{\mathscr{S}, \mathtt{x} : \mathtt{S} \vdash \mathtt{x} : \mathtt{S}}
\qquad
\frac{\text{[S-SNS]}}{\mathscr{S} \vdash s : \mathit{sort}(s)}
\qquad
\frac{\text{[S-GVAL]}}{\mathscr{S} \vdash \mathtt{g} : \mathit{sort}(\mathtt{g})}
$$

$$
\frac{\text{[S-PAIR]} \quad \mathscr{S} \vdash \mathtt{e}_1 : \mathtt{S}_1 \quad \mathscr{S} \vdash \mathtt{e}_2 : \mathtt{S}_2}{\mathscr{S} \vdash \texttt{<}\mathtt{e}_1,\mathtt{e}_2\texttt{>} : \texttt{<}\mathtt{S}_1,\mathtt{S}_2\texttt{>}}
\qquad
\frac{\text{[S-FST]} \quad \mathscr{S} \vdash \mathtt{e} : \texttt{<}\mathtt{S}_1,\mathtt{S}_2\texttt{>}}{\mathscr{S} \vdash \mathtt{fst}\ \mathtt{e} : \mathtt{S}_1}
\qquad
\frac{\text{[S-SND]} \quad \mathscr{S} \vdash \mathtt{e} : \texttt{<}\mathtt{S}_1,\mathtt{S}_2\texttt{>}}{\mathscr{S} \vdash \mathtt{snd}\ \mathtt{e} : \mathtt{S}_2}
$$

$$
\frac{\text{[S-COND]} \quad \mathscr{S} \vdash \mathtt{e}_0 : \mathtt{bool} \quad \mathscr{S} \vdash \mathtt{e}_1 : \mathtt{S}_1 \quad \mathscr{S} \vdash \mathtt{e}_2 : \mathtt{S}_2 \quad \mathtt{S} = \mathit{sup}_{\leq}(\mathtt{S}_1,\mathtt{S}_2)}{\mathscr{S} \vdash \mathtt{e}_0 ? \mathtt{e}_1 : \mathtt{e}_2 : \mathtt{S}}
$$

$$
\frac{\text{[S-COND-TRUE]} \quad \mathscr{S} \vdash \mathtt{e}_0 : \mathtt{true} \quad \mathscr{S} \vdash \mathtt{e}_1 : \mathtt{S}_1 \quad \mathscr{S} \vdash \mathtt{e}_2 : \mathtt{S}_2}{\mathscr{S} \vdash \mathtt{e}_0 ? \mathtt{e}_1 : \mathtt{e}_2 : \mathtt{S}_1}
$$

$$
\frac{\text{[S-COND-FALSE]} \quad \mathscr{S} \vdash \mathtt{e}_0 : \mathtt{false} \quad \mathscr{S} \vdash \mathtt{e}_1 : \mathtt{S}_1 \quad \mathscr{S} \vdash \mathtt{e}_2 : \mathtt{S}_2}{\mathscr{S} \vdash \mathtt{e}_0 ? \mathtt{e}_1 : \mathtt{e}_2 : \mathtt{S}_2}
$$

$$
\frac{\text{[S-FUN]} \quad \mathscr{S} \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \quad \mathtt{S}(\cdots) = \mathit{ms}(\textit{s-sigs}(\mathtt{f}),\overline{\mathtt{S}})}{\mathscr{S} \vdash \mathtt{f}(\overline{\mathtt{e}}) : \mathtt{S}}
$$

$$
\frac{\text{[S-SPR]} \quad \mathit{diffusion}(\mathtt{f}) \quad \mathscr{S} \vdash \mathtt{e}_0\overline{\mathtt{e}} : \mathtt{S}_0\overline{\mathtt{S}} \quad \mathtt{S}'(\cdots) = \mathit{ms}(\textit{stb-s-sigs}(\mathtt{f}),\mathtt{S}_0\overline{\mathtt{S}}) \quad \mathtt{S} = \mathit{sup}_{\leq}(\mathtt{S}_0,\mathtt{S}')}{\mathscr{S} \vdash \{\mathtt{e}_0 : \mathtt{f}(\mathtt{@},\overline{\mathtt{e}})\} : \mathtt{S}}
$$

User-defined function sort checking: $\boxed{\vdash \mathtt{D} : \overline{\mathtt{S}(\overline{\mathtt{S}})}}$

$$
\frac{\text{[S-DEF]} \quad \text{for all } \mathtt{S}(\overline{\mathtt{S}}) \in \textit{s-sigs}(\mathtt{d}), \quad \overline{\mathtt{x}} : \overline{\mathtt{S}} \vdash \mathtt{e} : \mathtt{S}' \quad \mathtt{S}' \leq \mathtt{S}}{\vdash \mathtt{def}\ \mathtt{T}\ \mathtt{d}(\overline{\mathtt{T}\ \mathtt{x}})\ \mathtt{is}\ \mathtt{e} : \textit{s-sigs}(\mathtt{d})}
$$

Figure 19: Sort-checking rules for expressions and function definitions

8.5. **Sort Checking.** In this section we present a decidable sort checking system for user-defined functions to check whether the sort-signature declarations provided by the mapping $\textit{s-sigs}(\cdot)$ are correct. The sort-checking rules are given in Figure 19. *Sort environments*, ranged over by $\mathscr{S}$ and written $\overline{\mathtt{x}} : \overline{\mathtt{S}}$, contain sort assumptions for program variables. The sort-checking judgement for expressions is of the form $\mathscr{S} \vdash \mathtt{e} : \mathtt{S}$, to be read: $\mathtt{e}$ has sort $\mathtt{S}$ under the sort assumptions $\mathscr{S}$ for the program variables occurring in $\mathtt{e}$. Sort checking of variables, sensors, ground values, pair constructions and deconstructions, and conditionals is similar to type checking. In particular, ground values and sensors are given a sort by construction by exploiting the mapping $\mathit{sort}(\cdot)$ introduced in Section 8.2, and the sort assigned to a conditional-expression is:

- the least upper bound $\mathit{sup}_{\leq}(\mathtt{S}_1,\mathtt{S}_2)$ of the sorts assigned to the branches (cf. Section 4.1) when the condition has sort `bool`;
- the sort assigned to the left branch when the condition has sort `true`; and
- the sort assigned to the right branch when the condition has sort `false`.

The sort-checking rule [S-FUN] for function application exploits the mapping $\textit{s-sigs}(\cdot)$ introduced in Section 8.3 and the auxiliary mapping $ms(\cdot,\cdot)$ introduced in Section 8.1. It first infers the sorts $\overline{\mathtt{S}}$ for the arguments $\overline{\mathtt{e}}$ of $\mathtt{f}$, then uses the most specific sort-signature for $\overline{\mathtt{S}}$ in $\textit{s-sigs}(\mathtt{f})$ for assigning to the application $\mathtt{f}(\overline{\mathtt{e}})$ the minimum sort $\mathtt{S}$ that can be assigned to $\mathtt{f}(\overline{\mathtt{e}})$ by using for $\mathtt{f}$ any of the sort signatures in $\textit{s-sigs}(\mathtt{f})$.

In a similar way, the sort-checking rule [S-SPR] for spreading expressions first infers the sorts $\mathtt{S}_0\overline{\mathtt{S}}$ for $\mathtt{e}_0\overline{\mathtt{e}}$, then retrieves the most specific sort-signature for $\mathtt{S}_0\overline{\mathtt{S}}$ in $\textit{stb-s-sigs}(\mathtt{f})$, $\mathtt{S}(\cdots)$, and finally assigns to the spreading expression the the least upper bound of $\mathtt{S}_0$ and $\mathtt{S}$.

The sort-checking rule for function definitions (which derives judgements of the form $\mathscr{S} \vdash \mathtt{D} : \overline{\mathtt{S}(\overline{\mathtt{S}})}$, where $\overline{\mathtt{S}(\overline{\mathtt{S}})} = \mathtt{S}^{(1)}(\overline{\mathtt{S}}^{(1)}),...,\mathtt{S}^{(n)}(\overline{\mathtt{S}}^{(n)})$ and $n \geq 1$) requires to check the definition $\mathtt{D}$ of a user-defined function $\mathtt{d}$ with respect to all the sort-signatures in $\textit{s-sigs}(\mathtt{d})$.

We say that a program (or library) **P** is *well sorted* to mean that all the user-defined function definitions in **P** sort check by using the rules in Figure 19.

Since no choice may be done when building a derivation for a given sort-checking judgment, the sort-checking rules straightforwardly describe a sort-checking algorithm.

Example 8.5. All user-defined functions provided in the examples in Sections 2.2 and 6.2 sort check by assuming the ground sorts and the subsorting given in Figure 8, the sorts for the ground values and sensors given in Figure 15, the sort-signatures for built-in functions given in Figure 16, the stabilising sort-signatures for built-in functions given in Figures 17 and the stabilising sort-signatures for user-defined diffusions given in 18.

8.6. **Sort Soundness of Device Computation.** In order to state the correctness of the sort-checking system presented in Section 8.5 we introduce the notion of set of well-sorted values trees for an expression, which generalizes to sorts the notion of set of well-typed values trees for an expression introduced in Section 5.1.

Given an expression $\mathtt{e}$ such that $\overline{\mathtt{x}} : \overline{\mathtt{S}} \vdash \mathtt{e} : \mathtt{S}$, the set $\textit{WSVT}(\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{e}, \mathtt{S})$ of the *well-sorted* value-trees for $\mathtt{e}$, is inductively defined as follows: $\theta \in \textit{WSVT}(\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{e}, \mathtt{S})$ if there exist

- a sensor mapping σ;
- well-formed tree environments $\overline{\theta} \in \textit{WSVT}(\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{e}, \mathtt{S})$; and
- values $\overline{\mathtt{v}}$ such that $\textit{length}(\overline{\mathtt{v}}) = \textit{length}(\overline{\mathtt{x}})$, $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{S}}'$ and $\overline{\mathtt{S}}' \leq \overline{\mathtt{S}}$;

such that $\sigma;\overline{\theta} \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] \Downarrow \theta$ holds—note that this definition is inductive, since the sequence of evaluation trees $\overline{\theta}$ may be empty.

The following theorem guarantees that from a properly sorted environment, evaluation of a well-sorted expression yields a properly sorted result.

Theorem 8.6 (Device computation sort preservation)**.** *If $\overline{\mathtt{x}} : \overline{\mathtt{S}} \vdash \mathtt{e} : \mathtt{S}$, σ is a sensor mapping, $\overline{\theta} \in \textit{WSVT}(\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{e}, \mathtt{S})$, $\textit{length}(\overline{\mathtt{v}}) = \textit{length}(\overline{\mathtt{x}})$, $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{S}}'$, $\overline{\mathtt{S}}' \leq \overline{\mathtt{S}}$, and $\sigma;\overline{\theta} \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] \Downarrow \theta$, then $\emptyset \vdash \rho(\theta) : \mathtt{S}'$ for some $\mathtt{S}'$ such that $\mathtt{S}' \leq \mathtt{S}$.*

Proof. See Appendix C. □

Remark 8.7 (On the relation between type checking and sort checking)**.** A sort system should be such that all the programs (or libraries) accepted by the sort system are accepted by the original type system and vice-versa (cf. the discussion at the beginning of Section 4.1). However, the sort system considered in this paper has a peculiarity: it checks that every diffusion-expressions $\{\mathtt{e}_1 : \mathtt{f}(@, \mathtt{e}_2, ..., \mathtt{e}_n)\}$ occurring in **P** is sort-checked by considering for $\mathtt{f}$ only the sort-signatures in $\textit{stb-s-sigs}(\mathtt{f})$—which

is assumed to be such that for all $\mathtt{S}(\overline{\mathtt{S}}) \in$ *stb-s-sigs*$(\mathtt{f})$ the predicate *stabilising*$(\mathtt{f},\mathtt{S}(\overline{\mathtt{S}}))$ holds. Therefore, some well-typed programs (or libraries)—including all the non-self-stabilising programs (like the one considered in Example 3.8)—do not sort check.

The standard relation between the sort system and the type system that it refines holds for programs (or libraries) that do not contain spreading expressions. I.e., all the programs (or libraries) that do not contain spreading expressions that accepted by the original type system are accepted by the sort system and vice-versa. In particular, whenever all the sorts are trivial (i.e., $\mathbf{sorts}(\mathtt{T}) = \{\mathtt{T}\}$, for every type $\mathtt{T}$) we have that, on programs (or libraries) that do not contain spreading expressions, the sort checking rules (in Figure 19) behave exactly as the type-checking rules (in Figure 4).

9. Checking Stabilising Assumptions for User-Defined Diffusions

In this section we present a decidable annotated sort checking system that guarantees that if a program **P** is *well-sorted* (i.e., it can be successfully checked by the rules of the system given in Section 8 and, therefore, satisfies Condition (1) of Section 4.3) guarantees that, under the additional requirements (1) and (2) introduced at the beginning of Section 7.1.2, if **P** is *well-annotated* (i.e., it can be successfully checked by the rules of the system) then Condition (2) of Section 4.3 holds, and hence self-stabilisation follows.

To this aim, under the additional requirements (1) and (2) introduced at the beginning of Section 7, we assume that each program **P** comes with a mapping *a-s-sigs*$(\cdot)$ that associates to each diffusion $\mathtt{f}$ a (possibly empty) set of annotated sort-signatures *a-s-sigs*$(\mathtt{f})$ (we write π-*s-sigs*$(\cdot)$ to denote the mapping such that π-*s-sigs*$(\mathtt{f}) = \{\mathtt{S}(\overline{\mathtt{S}})[\pi] \mid \mathtt{S}(\overline{\mathtt{S}})[\pi] \in$ *a-s-sigs*$(\mathtt{f})\}$) such that the following conditions are satisfied:

- for every diffusion $\mathtt{f}$ of type $\mathtt{T}(\cdots)$, if $\mathtt{T}$ is ground, then *stb-s-sigs*$(\mathtt{f}) = |$*!-s-sigs*$(\mathtt{f})|$;
- for every user-defined diffusion $\mathtt{d}$ of the form displayed in Equation 7.1 of Section 7.2, consider the function $\mathtt{f}$ occurring in the body of $\mathtt{d}$:
  - if $\mathtt{f}$ is built-in function, then *stb-s-sigs*$(\mathtt{d}) \subseteq |$*!-s-sigs*$(\mathtt{f})|$, and
  - if $\mathtt{f}$ is user-defined, then *stb-s-sigs*$(\mathtt{d}) = |$*!-s-sigs*$(\mathtt{f})|$.

Note that the above conditions (which can be checked automatically) imply that, for every user defined function $\mathtt{d}$, the value of *stb-s-sigs*$(\mathtt{d})$ is completely defined by the mapping *a-s-sigs*$(\cdot)$—therefore, there is no need to explicitly define the value of *stb-s-sigs*$(\cdot)$ for user-defined diffusions. The assumptions *a-s-sigs*$(\mathtt{b})$ for the built-in functions $\mathtt{b}$ are considered valid—they should come with the definition of the language. Instead, the validity of the assumptions *a-s-sigs*$(\mathtt{d})$ for the user-defined functions $\mathtt{d}$ must be checked—these assumptions could be either (possibly partially) provided by the user or automatically inferred.[8] Therefore, in order to check that Condition (2) of Section 4.3 holds, it is enough to check that each the user-defined diffusions $\mathtt{d}$ of **P** has all the annotated sort signatures *a-s-sigs*$(\mathtt{d})$.

We first introduce some auxiliary definitions (in Section 9.1), then we consider the issue of associating annotated sort-signatures to diffusions (in Section 9.2) and the issue of associating annotated sorts to values (in Section 9.3), and finally we present a decidable annotated sort checking system for checking the correctness of annotated sort-signature declarations for user-defined diffusions (in Section 9.4) and show its soundness (in Section 9.5).

---

[8]The naive inference approach, that is: inferring *a-s-sigs*$(\mathtt{d})$ by checking all the sort-signature of $\mathtt{d}$ is linear in the number of elements of *s-sigs*$(\mathtt{d})$. Some optimizations are possible. We do not address this issue in the paper.

9.1. **Auxiliary definitions.** In this section we adapt the notions of minimal set of sort-signatures, deterministic set of sort signatures and most specific sort-signature (cf. Section 8.1) to annotated sort-signatures. These notions will be used (in Section 9.4) to formulate the annotated sort-checking rules for function applications.

Given a diffusion signature $\mathtt{T}(\mathtt{T}\overline{\mathtt{T}})$ and a set of annotated sort-signatures $Q \subseteq$ **annotated-sort-signatures**$(\mathtt{T}(\mathtt{T}\overline{\mathtt{T}}))$, an annotated sort $\mathtt{S}'_1\,[\pi']$ such that $\mathtt{S}'_1 \in$ **sort-signatures**$(\mathtt{T}_1)$ and some sorts $\overline{\mathtt{S}}' \in$ **sort-signatures**$(\overline{\mathtt{T}})$, consider the (possibly empty) subset of Q defined as follows:

$$Q(\mathtt{S}'_1\,[\pi']\,\overline{\mathtt{S}}') = \{\mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}})\,[\pi''] \in Q \mid \mathit{key}(\mathtt{S}'_1) \leq^{\text{progressive}} \mathit{key}(\mathtt{S}_1),\ \ \mathtt{S}'_1 \leq \mathtt{S}_1 \text{ and } \overline{\mathtt{S}}' \leq \overline{\mathtt{S}}\}.$$

We say that Q is *deterministic*, notation *deterministic*(Q), to mean that for all $\mathtt{S}'_1\,[\pi']\,\overline{\mathtt{S}}'$ there exists an annotated sort-signature $\mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}})\,[\pi] \in Q(\mathtt{S}'_1\,[\pi']\,\overline{\mathtt{S}}')$, called the *most specific annotated sort-signature for* $\overline{\mathtt{S}}'$ *in* Q, such that:

$$\text{for all } \mathtt{S}''(\overline{\mathtt{S}}'')\,[\pi''] \in Q(\mathtt{S}'_1\,[\pi']\,\overline{\mathtt{S}}') \text{ it holds that } \mathtt{S}\,[\pi(\pi')] \leq \mathtt{S}''\,[\pi''(\pi')].$$

The mapping $\mathit{ms}(Q, \mathtt{S}'_1\,[\pi']\,\overline{\mathtt{S}}')$, given a deterministic set of sort-signatures $Q \subseteq$ **annotated-sort-signatures**$(\mathtt{T}(\mathtt{T}\overline{\mathtt{T}}))$, an annotated sort $\mathtt{S}'_1\,[\pi']$ such that $\mathtt{S}'_1 \in$ **sort-signatures**$(\mathtt{T}_1)$ and some sorts $\overline{\mathtt{S}}' \in$ **sort-signatures**$(\overline{\mathtt{T}})$, returns the *most specific annotated sort-signature for* $\mathtt{S}'_1\,[\pi']\,\overline{\mathtt{S}}'$ *in* Q if $Q(\mathtt{S}'_1\,[\pi']\,\overline{\mathtt{S}}')$ is not empty, and is undefined otherwise.

9.2. **Annotated Sort-Signatures for Diffusions.** The mapping *a-s-sigs*$(\cdot)$ and the conditions that provides its link with the mapping *stb-s-sigs*$()$ have been illustrated at the beginning of Section 9. Here, we illustrate some additional condition that is needed to simplify the formulation of the annotated sort checking rules and to guarantee their soundness.

We assume that for each built-in diffusion $\mathtt{b}$ the (possibly empty) set of annotated sort-signatures *a-s-sigs*$(\mathtt{b})$ is such that the following conditions are satisfied:

- *a-s-sigs*$(\mathtt{b})$ is minimized and deterministic;
- *a-s-sigs*$(\mathtt{b}) \subseteq$ **annotated-sort-signatures**$(\mathtt{b})$; and
- *a-s-sigs*$(\mathtt{b})$ represents all the annotated sort-signatures satisfied by $\mathtt{b}$, i.e., for each $\mathtt{S}'(\overline{\mathtt{S}}')\,[\pi'] \in$ **annotated-sort-signatures**$(\mathtt{b})$ there exists $\mathtt{S}(\overline{\mathtt{S}})\,[\pi] \in$ *a-s-sigs*$(\mathtt{b})$ such that $\mathtt{S}(\overline{\mathtt{S}})\,[\pi] \leq \mathtt{S}'(\overline{\mathtt{S}}')\,[\pi']$ holds.

Note that the first of the above three conditions on the mapping *a-s-sigs*$(\mathtt{b})$ can be checked automatically.

Example 9.1. Figure 20 illustrates the annotated sort-signatures for the built-in diffusions used in the examples introduced thought the paper.

We also assume that for each user-defined diffusion $\mathtt{d}$ the (possibly empty) set of annotated sort-signatures *a-s-sigs*$(\mathtt{d})$ is minimized and deterministic (note that this condition can be checked automatically).

Example 9.2. Figure 21 gives minimal deterministic sets of annotated sort signatures for the user-defined π-prestabilising diffusions that allow to successfully check the user-defined diffusions used in the examples introduced thought the paper.

$$\begin{array}{rcl}
\textit{a-s-sigs}(\texttt{or}) & = & \texttt{false}(\texttt{false},\texttt{false})\texttt{[!]},\\
& & \texttt{true}(\texttt{true},\texttt{bool})\texttt{[!]},\\
& & \texttt{true}(\texttt{bool},\texttt{true})\texttt{[!]}\\[1em]
\textit{a-s-sigs}(+) & = & \texttt{nr}(\texttt{nr},\texttt{zr})\texttt{[?]},\\
& & \texttt{znr}(\texttt{znr},\texttt{zr})\texttt{[?]},\\
& & \texttt{zr}(\texttt{zr},\texttt{zr})\texttt{[!]},\\
& & \texttt{zpr}(\texttt{zpr},\texttt{zpr})\texttt{[?]},\\
& & \texttt{pr}(\texttt{zpr},\texttt{pr})\texttt{[!]},\\
& & \texttt{pr}(\texttt{pr},\texttt{zpr})\texttt{[?]},\\
& & \texttt{real}(\texttt{real},\texttt{zpr})\texttt{[?]},\\
& & \texttt{real}(\texttt{real},\texttt{pr})\texttt{[!]}
\end{array}$$

Figure 20: Annotated sort signatures for built-in π-prestabilising diffusions used in the examples (cf. Figure 16)

$$\begin{array}{rcl}
\textit{a-s-sigs}(\texttt{restrict}) & = & \texttt{real}(\texttt{real},\texttt{bool})\texttt{[?]}\\
\textit{a-s-sigs}(\texttt{restrictSum}) & = & \texttt{real}(\texttt{real},\texttt{pr},\texttt{bool})\texttt{[!]}\\
\textit{a-s-sigs}(\texttt{sum_or}) & = & \texttt{<real,bool>}(\texttt{<real,bool>},\texttt{<pr,bool>})\texttt{[!]}\\
\textit{a-s-sigs}(\texttt{add_to_1st}) & = & \texttt{<real,real>}(\texttt{<real,real>},\texttt{pr})\texttt{[!]}
\end{array}$$

Figure 21: Annotated sort signatures for the user-defined π-prestabilising diffusions used in the examples (cf. Figure 18)

$$\begin{array}{rcl}
\textit{a-sort}(\texttt{FALSE}) & = & \texttt{false[!]}\\
\textit{a-sort}(\texttt{TRUE}) & = & \texttt{true[!]}\\
\textit{a-sort}(0) & = & \texttt{zr[!]}\\
\textit{a-sort}(\texttt{POSINF}) & = & \texttt{pr[!]}
\end{array}$$

Figure 22: Annotated sorts for the ground values used in the examples

9.3. **Annotated Sorts for Values.** We assume a partial mapping $\textit{a-sort}(\cdot)$ that for each ground value g:

- returns the annotated sort $\textit{sort}(\texttt{g})\texttt{[!]}$, if g is the maximum element of $[\![\textit{sort}(\texttt{g})]\!]$ w.r.t. $\leq_{\textit{type}(\texttt{g})}$, and
- is undefined, otherwise.

Note that Proposition 7.5 guarantees the soundness of the mapping $\textit{a-sort}(\cdot)$.

Example 9.3. Figure 22 illustrates the `!`-annotated sorts for the ground values used in the examples introduced thought the paper.

9.4. **Annotated Sort Checking for User-Defined Diffusions.** In this section we present a decidable annotated sort checking system to check whether the π-annotated sort-signature assumptions for the user-defined diffusions d provided by the mapping $\textit{a-s-sigs}(\cdot)$ are correct. The annotated sort-checking rules are given in Figure 23.

Pure expression annotated sort checking: $\boxed{\mathscr{A} \vdash \mathtt{e} : \mathtt{A}}$

$$\text{[A-VAR]}\ \frac{}{\mathscr{A}, \mathtt{x} : \mathtt{S[?]} \vdash \mathtt{x} : \mathtt{S[?]}} \qquad \text{[A-GVAL]}\ \frac{\top_{key(\mathtt{S})} = \mathtt{g}}{\mathscr{A}, \mathtt{x} : \mathtt{S[?]} \vdash \mathtt{g} : \textit{a-sort}(\mathtt{g})}$$

$$\text{[A-PAIR]}\ \frac{\mathscr{A} \vdash \mathtt{e}_1 : \mathtt{S}_1\mathtt{[}\pi\mathtt{]} \quad |\mathscr{A}| \vdash \mathtt{e}_2 : \mathtt{S}_2}{\mathscr{A} \vdash \mathtt{<e_1,e_2>} : \mathtt{<S_1,S_2>[}\pi\mathtt{]}} \qquad \text{[A-FST]}\ \frac{\mathscr{A} \vdash \mathtt{e} : \mathtt{<S_1,S_2>[}\pi\mathtt{]}}{\mathscr{A} \vdash \mathtt{fst}\ \mathtt{e} : \mathtt{S}_1\mathtt{[}\pi\mathtt{]}}$$

$$\text{[A-COND]}\ \frac{|\mathscr{A}| \vdash \mathtt{e}_0 : \mathtt{bool} \quad \mathscr{A} \vdash \mathtt{e}_1 : \mathtt{A}_1 \quad \mathscr{A} \vdash \mathtt{e}_2 : \mathtt{A}_2 \quad \mathtt{A} = sup_{\leq}(\mathtt{A}_1, \mathtt{A}_2)}{\mathscr{A} \vdash \mathtt{e}_0\mathtt{?e}_1\mathtt{:e}_2 : \mathtt{A}}$$

$$\text{[A-COND-TRUE]}\ \frac{|\mathscr{A}| \vdash \mathtt{e}_0 : \mathtt{true} \quad \mathscr{A} \vdash \mathtt{e}_1 : \mathtt{A} \quad |\mathscr{A}| \vdash \mathtt{e}_2 : \mathtt{S}}{\mathscr{A} \vdash \mathtt{e}_0\mathtt{?e}_1\mathtt{:e}_2 : \mathtt{A}}$$

$$\text{[A-COND-FALSE]}\ \frac{|\mathscr{A}| \vdash \mathtt{e}_0 : \mathtt{false} \quad |\mathscr{A}| \vdash \mathtt{e}_1 : \mathtt{S} \quad \mathscr{A} \vdash \mathtt{e}_2 : \mathtt{A}}{\mathscr{A} \vdash \mathtt{e}_0\mathtt{?e}_1\mathtt{:e}_2 : \mathtt{A}}$$

$$\text{[A-FUN]}\ \frac{\mathscr{A} \vdash \mathtt{e}_1 : \mathtt{S}_1\mathtt{[}\pi''\mathtt{]} \qquad |\mathscr{A}| \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \mathtt{S}(\cdots)\mathtt{[}\pi'\mathtt{]} \in ms(\textit{a-s-sigs}(\mathtt{f}), \mathtt{S}_1\overline{\mathtt{S}}) \qquad \pi = \pi'(\pi'')}{\mathscr{A} \vdash \mathtt{f}(\mathtt{e}_1, \overline{\mathtt{e}}) : \mathtt{S[}\pi\mathtt{]}}$$

User-defined diffusion annotated sort checking: $\boxed{\vdash \mathtt{D} : \overline{\mathtt{S}(\overline{\mathtt{S}})\mathtt{[}\pi\mathtt{]}}}$

$$\text{[A-DEF]}\ \frac{\begin{array}{l}\text{for all } \mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}})\mathtt{[}\pi\mathtt{]} \in \textit{a-s-sigs}(\mathtt{d}),\\ \quad \overline{\mathtt{x}} : \mathtt{S}_1\mathtt{[?]}\,\overline{\mathtt{S}} \vdash \mathtt{e} : \mathtt{S}'\mathtt{[}\pi'\mathtt{]} \qquad \mathtt{S}'\mathtt{[}\pi'\mathtt{]} \leq \mathtt{S[}\pi\mathtt{]}\end{array}}{\vdash \mathtt{def\ T\ d}(\overline{\mathtt{T\ x}})\ \mathtt{is\ e} : \textit{a-s-sigs}(\mathtt{d})}$$

Figure 23: Annotated sort checking rules for expressions and diffusion definitions

The check a user defined diffusion $\mathtt{d}$ has the annotated sort signature $(\mathtt{S}_1, ..., \mathtt{S}_n)\mathtt{S[}\pi\mathtt{]}$ can be done by assuming annotated sort $\mathtt{S}_1\mathtt{[?]}$ for the first formal parameter of $\mathtt{d}$, assuming sorts $\mathtt{S}_2, ..., \mathtt{S}_n$ for the other formal parameters of $\mathtt{d}$, and trying to assign to the body of $\mathtt{d}$ an annotated sort $\mathtt{S}'\mathtt{[}\pi'\mathtt{]}$ such that $\mathtt{S}'\mathtt{[}\pi'\mathtt{]} \leq \mathtt{S[}\pi\mathtt{]}$. According to this observation we introduce the notion of *annotated sort environments*, ranged over by $\mathscr{A}$ and written $\mathtt{x} : \mathtt{S[?]}, \overline{\mathtt{x}} : \overline{\mathtt{S}}$, that contain one ?-annotated sort assumption and some (possibly none) sort assumptions for program variables. The annotated sort checking rule for user-defined diffusions [A-DEF] (which derives judgements of the form $\vdash \mathtt{D} : \overline{\mathtt{S}(\overline{\mathtt{S}})\mathtt{[}\pi\mathtt{]}}$) uses this strategy to check that the definition of a user-defined diffusion $\mathtt{d}$ with respect to all the annotated sort signatures in $\textit{a-s-sigs}(\mathtt{d})$.

The annotated sort-checking judgement for expressions is of the form $\mathscr{A} \vdash \mathtt{e} : \mathtt{A}$, to be read: pure-expression $\mathtt{e}$ has annotated sort $\mathtt{A}$ under the assumptions $\mathscr{A}$ for the program variables occurring in $\mathtt{e}$. The support of an annotated sort environment $\mathscr{A}$, denoted by $|\mathscr{A}|$, is the sort environment obtained from $\mathscr{A}$ by removing the input annotation, i.e.,

$$|\mathtt{x} : \mathtt{S[?]}, \overline{\mathtt{x}} : \overline{\mathtt{S}}| = \mathtt{x} : \mathtt{S}, \overline{\mathtt{x}} : \overline{\mathtt{S}}$$

(cf. Section 7.4). Some of the annotated sort-checking rules rely on the judgements of the sort checking system introduced in Section 9.2 to sort check some subexpressions. Namely: the right element of the pair in rule [A-PAIR]; the condition of the conditional-expression in rules [A-COND], [A-COND-TRUE] and [A-COND-FALSE]; the right branch of the conditional-expression in rule [A-COND-TRUE];

the left branch of the conditional-expression in rule [A-COND-FALSE]; and the arguments (excluding the first one) of the pure function $\mathtt{f}$ (that must be an π-diffusion) in rule [A-FUN].

Annotated sort checking of variables and ground values is similar to sort checking. In particular, ground values may be given an annotated sort by construction by exploiting the mapping *a-sort*$(\cdot)$—the premise $\top_{key(\mathtt{S})} = \mathtt{g}$ ensures that the value $\mathtt{g}$ is relevant to the overall goal of the annotated sort checking derivation process, that is: deriving an annotated sort $\mathtt{S}'[\pi']$ such that $\mathtt{S}'[\pi'] \leq \mathtt{S}[\pi]$ for the body $\mathtt{e}$ of a user-defined diffusion $\mathtt{d}$ in order to check that $\mathtt{d}$ has the annotated sort-signature $\mathtt{S}(\mathtt{S}_1,...,\mathtt{S}_n)[\pi]$.

Note that there is no annotated sort checking rule for expressions of the form $\mathtt{snd}\ \mathtt{e}$—because of the leftmost-as-key preorder such an expression is not relevant to the overall goal of the annotated sort checking derivation process.

The sort-checking rule [A-FUN] for diffusion application exploits the mapping *a-s-sigs*$(\cdot)$ and the auxiliary mapping *ms*$(\cdot,\cdot)$ introduced in Section 9.1. It first infers the sort $\mathtt{S}_1[\pi'']$ for the first argument $\mathtt{e}_1$ of $\mathtt{f}$ and the sorts $\overline{\mathtt{S}}$ for remaining the arguments $\overline{\mathtt{e}}$ of $\mathtt{f}$, then uses the most specific annotated sort signature for $\overline{\mathtt{S}}$ in *a-s-sigs*$(\mathtt{f})$ for assigning to the application $\mathtt{f}(\overline{\mathtt{e}})$ the minimum annotated sort $\mathtt{S}[\pi']$ that can be assigned to $\mathtt{f}(\overline{\mathtt{e}})$ by using for $\mathtt{f}$ any of the annotated sort signatures in *s-sigs*$(\mathtt{f})$.

We say that a program (or library) **P** is *well annotated* to mean that: all the user-defined function definitions in **P** check by using the sort-checking rules in Figure 19, and all the user-defined diffusion definitions sort checking rules in Figure 23.

Example 9.4. All user-defined functions provided in the examples in Sections 2.2 and 6.2 sort-check and (when they are diffusions) annotate sort check by assuming the ground sorts and the subsorting given in Figure 8, the sorts for the ground values and sensors given in Figure 15, the sort-signatures for built-in functions given in Figure 16, the annotated sorts for ground values given in Figure 22, the annotated sort-signatures for built-in functions given in Figures 20 and the annotated sort-signatures for user-defined diffusions given in 21.

Since no choice may be done when building a derivation for a given annotated sort-checking judgment, the annotated sort-checking rules straightforwardly describe an annotated sort-checking algorithm.

9.5. **Annotation Soundness.** The following theorem states the correctness of the annotation-checking system presented in Section 9.4

Theorem 9.5 (Annotation soundness)**.** If $\vdash \mathtt{D} : \overline{\mathtt{S}(\overline{\mathtt{S}})[\pi]}$ holds, then π-*prestabilising*$(\mathtt{f},\mathtt{S}(\overline{\mathtt{S}}))$ holds for all $\mathtt{S}(\overline{\mathtt{S}})[\pi] \in \overline{\mathtt{S}(\overline{\mathtt{S}})[\pi]}$.

Proof. See Appendix D. □

10. Related Work and Discussion

We here discuss the main related pieces of work, rooted in previous research on finding core models for spatial computing and self-organisation, formal approaches for large-scale systems, and finally on self-stabilisation in distributed systems.

10.1. **Spatial computing and self-organisation.** A first step in studying general behavioural properties of self-organising systems is the identification of a reference model, making it possible to reuse results across many different models, languages and platforms. The review in [7] surveys a good deal of the approaches considering some notion of space-time computations, which are the basis for any self-organising system. Examples of such models include the Hood sensor network abstraction [50], the $\sigma\tau$-Linda model [47], the SAPERE computing model [48], and TOTA middleware [35], which all implement computational fields using similar notions of spreading. More generally, Proto [36, 5] and its core formalisation as the "field calculus" [16], provides a functional model that appear general enough to serve as a starting point for investigating behavioural aspects of spatial computation and self-organisation [9]. In fact, in [10] it is proved that the field calculus is universal, in the sense that it can be used to describe any causal and discretely-approximable computation in space-time.

Hence, we started from the field calculus, in which computation is expressed by the functional combination of input fields (as provided by sensors), combined with mechanisms of space-based (neighbour) data aggregation, restriction (distributed branch) and state persistence. The calculus presented here is a fragment of the field calculus, focussing on only two basic computational elements: *(i)* functional composition of fields, and *(ii)* a spreading expression. In particular, the latter is a suitable combination of basic mechanisms of the field calculus, for which we were able to prove convergence to a single final state. Namely, a spreading expression $\{\mathtt{e}_0 : \mathtt{g}(\mathtt{@},\mathtt{e}_1,..,\mathtt{e}_n)\}$ in our calculus is equivalent to the following field calculus expression:

$$\mathtt{(rep\ x\ (inf)\ (min\ e_0\ (g\ (min\text{-}hood+\ (nbr\ x))\ e_1\ ..\ \ e_n)))}$$

In particular, it was key to our end to neglect recursive function calls (in order to ensure termination of device fires, since the calculus does not model the domain restriction construct [46, 36] and both the branches of a conditional expression are evaluated), stateful operations (in our model, the state of a device is always cleaned up before computing the new one), and to restrict aggregation to minimum function and progression to what we called "stabilising diffusion" functions.

Other than applying to fragments of the field calculus, the result provided here can be applied to rule-based systems like those of the SAPERE approach [48] and of rewrite-based coordination models [47], along the lines depicted in [43]. Note that our condition for self-stabilisation is only a sufficient one. A primary example of the fact that it is not necessary is Laplacian consensus [24], expressed as follows in the field calculus:

$$\mathtt{(rep\ x\ e_i\ (+\ x\ (*\ e_\varepsilon\ (sum\text{-}hood\ (-\ (nbr\ x)\ x))))))}$$

It cannot be expressed in the calculus we propose here, but still stabilises to a plateau field, computed as a consensus among the values of input field $\mathtt{e}_i$ (with $\mathtt{e}_\varepsilon$) driving the dynamics of the output field. Other cases include so-called convergence cast [8].

10.2. **Formal approaches.** In this paper we are interested in formally predicting the behaviour of a complex system, in which the local interactions among a possibly miriad of devices make a global and robust pattern of behaviour emerge. In the general case, one such kind of prediction can hardly be obtained.

The quintessential formal approach, model-checking [14], cannot typically scale with the number of involved components: suitable abstractions are needed to model arbitrary-size systems (as in [18]), which however only work in very constrained situations. Approximate model-checking [29, 12], basically consisting in a high number of simulation bursts, is viable in principle, but it still falls under the umbrella of semi-empirical evaluations, for only statistical results are provided. Recently, fluid flow approximation has been proposed to turn large-scale computational systems into systems

of differential equations that one could solve analytically or use to derive an evaluation of system behaviour [11]. Unfortunately, this approach seems developed yet only to abstract from the number of (equivalent and non-situated) agents performing a repetitive task, instead of abstracting from the discreteness of a large-scale situated computational network.

Recent works finally aim at proving properties of large-scale systems by hand-written proofs, which are the works most related to the result of present paper. The only work aiming at a mathematical proof of stabilisation for the specific case of computational fields is [6]. There, a self-healing gradient algorithm called CRF (constraints and restoring forces) is introduced to estimate physical distance in a spatial computer, where the neighbouring relation is fixed to unit-disc radio, and node firing is strictly connected to physical time. Compared to our approach, the work in [6] tackles a more specific problem, and is highly dependent on the underlying spatial computer assumptions. Another work presenting a proof methodology that could be helpful in future stages of our research is the universality study in [10].

From the viewpoint of rewrite semantics [31], which is the meta-model closest to our formalisation attempt, our proof is most closely related to the *confluence* property, that is, don't care non-determinism. Our result entails confluence, but it is actually a much strongest property of global uniqueness of a normal form, independently of initial state.

10.3. **Self-stabilisation.** Our work concerns the problem of identifying complex network computations whose outcome is predictable. The notion we focus on requires a unique global state being reached in finite time independently of the initial state, that is, depending only on the state of the environment (topology and sensors). It is named *(strong) self-stabilisation* since it is related with a usual notion of self-stabilisation to *correct* states for distributed systems [22], defined in terms of a set C of correct states in which the system eventually enters in finite time, and then never escapes from – in our case, C is made by the single state corresponding to the sougth result of computation.

Actually many different versions of the notion of self-stabilisation have been adopted in past, surveyed in [41], from works of Dijkstra's [20, 19] to more recent and abstract ones [1], typically depending on the reference model for the system to study—protocols, state machines. In our case, self-stabilisation is studied for a distributed data structure (the computational field). Previous work on this context like [30] however only considers the case of heap-like data structures in a non-distributed settings: this generally makes it difficult to draw a bridge with existing research.

Several variations of the definition also deal with different levels of quality (fairness, performance). For instance, the notion of superstabilisation [23] adds to the standard self-stabilisation definition a requirement on a "passage predicate" that should hold while a system recovers from a specific topological change. Our work does not address this very issue, since we currently completely equate the treatment of topological changes and changes to the inputs (i.e., sensors), and do not address specific performance requirements. However, future works addressing performance issues will likely require some of the techniques studied in [23]. Performance is also affected by the fairness assumption adopted: we relied on a notion abstracting from more concrete ones typically used [33], which we could use as well though losing a bit of the generality of our result.

Concerning the specific technical result achieved here, the closest one appears to be the creation of a hop-count gradient, which is known to self-stabilise: this is used in [22] as a preliminary step in the creation of the spanning tree of a graph. The main novelty in this context is that self-stabilisation is not proved here for a specific algorithm/system: it is proved for all fields inductively obtained by functional composition of fixed fields (sensors, values) and by a gradient-inspired spreading process. Other works attempt to devise general methodologies for building self-stabilising systems like we do. The work in [2] depicts a compiler turning any protocol into a self-stabilising one.

Though this is technically unrelated to our solution, it shares the philosophy of hiding the details of how self-stabilisation is achieved under the hood of the execution platform: in our case in fact, the designer wants to focus on the macro-level specification, trusting that components behave and interact so as to achieve the global outcome in a self-stabilising way. The work in [28] suggests that hierarchical composition of self-stabilising programs is self-stabilising: an idea that is key to the construction of a functional language of self-stabilising "programs".

In spite of the connection with some of these previous works, to the best of our knowledge ours is novel under different dimensions. First, it is the first attempt of providing a notion of self-stabilisation directly connected to the problem of engineering self-organisation. Secondly, the idea of appliying it to a whole specification language is also new, along with the fact that we apply a type-based approach, providing a correct checking procedure that paves the way towards compiler support. As we use now a type-based approach, other static analysis techniques may be worth studying in future attempts (see, e.g., [39]).

11. Conclusions and Future Work

Emerging application scenarios like pervasive computing, robotic systems, and wireless sensor networks call for developing robust and predictable large-scale situated systems. However, the diffusion/aggregation processes that are typically to be implemented therein are source of complex phenomena, and are notoriously very hard to be formally treated. The goal of this work is to bootstrap a research thread in which mechanisms of self-organisation are captured by linguistic constructs, so that static analysis in the programming language style can be used to isolate fragments with provable predictable behaviour. In the medium term, we believe this is key to provide a tool-chain (programming language, libraries, simulation and execution platforms) which enables the development of complex software systems whose behaviour has still some predictability obtained "by construction".

Along this line, this paper studies a notion of strong self-stabilisation, identifying a sufficient condition expressed on the diffusion/aggregation mechanisms occurring in the system. This targets the remarkable situation in which the final shape of the distributed data structure created (i.e., the computational field) is deterministically established, does not depend on transient events (such as temporaneous failures), and is only determined by the stabilised network topology. Namely, this is the case in which we can associate to a complex computation a deterministic and easily computable result.

It would be interesting to relax some of the conditions and assumptions we relied upon in this paper, so as to provide a more general self-stabilisation result. First of all, the current definition of self-stabilisation requires values to have upper-bounds, to prevent network subparts that become isolated from "sources" (e.g., of a gradient) to be associated with values that grow to infinity without reaching a fixpoint: a more involved definition of self-stabilisation could be given that declares such a divergence as admitted, allowing us to relax "noetherianity" of values. We also plan to extend the result to encompass the domain restriction construct [46, 36] (i.e., to add to the calculus a form of conditional expression where only one of the branches is evaluated). In this way also recursive function definitions could be added (then, in order to guarantee termination of computation rounds, standard analysis techniques for checking termination of recursive function definitions might be used). We currently focus only on spreading-like self-stabilisation, whether recent works [8] suggest that "aggregation" patterns can be similarly addressed as well, though they might require a completely different language and proof methodology.

Of course, other behavioural properties are of interest which we plan to study in future work as an extension to the results discussed here. First, it is key to study notions of self-stabilisation for computational fields which are designed so as to be dynamically evolving, like e.g. the anticipative gradient [37]. Second, it would be interesting to extend our notion of self-stabilisation so as to take into account those cases in which only approximate reachability of the sought state is required, since it can lead to computations with better average performance, as proposed in [4]. Other aspects of interest that can be formally handled include performance characterisation, code mobility, expressiveness of mechanisms, and independence of network density, which will likely be subject of next investigations as well.

Acknowledgement

We thank the anonymous COORDINATION 2014 referees for useful comments on an earlier version of this paper, and the anonymous LMCS referees for insightful comments and suggestions for improving the presentation.

References

[1] A. Arora and M. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.

[2] B. Awerbuch and G. Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 258–267, 1991.

[3] J. Bachrach, J. Beal, and J. McLurkin. Composable continuous space programs for robotic swarms. *Neural Computing and Applications*, 19(6):825–847, 2010.

[4] J. Beal. Flexible self-healing gradients. In S. Y. Shin and S. Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 1197–1201. ACM, 2009.

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

[6] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin. Fast self-healing gradients. In *Proceedings of ACM SAC 2008*, pages 1969–1975. ACM, 2008.

[7] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013. A longer version available at: `http://arxiv.org/abs/1202.5509`.

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

[9] J. Beal and M. Viroli. Space–time programming. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 373(2046), 2015.

[10] J. Beal, M. Viroli, and F. Damiani. Towards a unified model of spatial computing. In *7th Spatial Computing Workshop (SCW 2014)*, AAMAS 2014, Paris, France, May 2014.

[11] L. Bortolussi, D. Latella, and M. Massink. Stochastic process algebra and stability analysis of collective systems. In R. D. Nicola and C. Julien, editors, *Coordination Models and Languages, 15th International Conference, COORDINATION 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, volume 7890 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.

[12] M. Casadei and M. Viroli. Toward approximate stochastic model checking of computational fields for pervasive computing systems. In J. Pitt, editor, *Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, pages 199–204. IEEE CS, Apr. 2012. 2012 IEEE Sixth International Conference (SASOW 2012), Lyon, France, 10-14 Sept. 2012. Proceedings.

[13] R. Claes, T. Holvoet, and D. Weyns. A decentralized approach for anticipatory vehicle routing using delegate multiagent systems. *IEEE Transactions on Intelligent Transportation Systems*, 12(2):364–373, 2011.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, January 2000.
[15] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, June 1996.
[16] F. Damiani, M. Viroli, D. Pianini, and J. Beal. Code mobility meets self-organisation: A higher-order calculus of computational fields. In S. Graf and M. Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 9039 of *Lecture Notes in Computer Science*, pages 113–128. Springer International Publishing, 2015.
[17] R. Davies. *Practical Refinement-Type Checking*. PhD thesis, CMU, Pittsburgh, PA, USA, 2005.
[18] G. Delzanno. An overview of msr(c): A clp-based framework for the symbolic verification of parameterized concurrent systems. *Electronic Notes in Theoretical Computer Science*, 76, 2002.
[19] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17(11):643–644, 1974.
[20] E. Dijkstra. Ewd391 self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective*, pages 41–46. Springer-Verlag, 1982. EWD391's original date is 1973.
[21] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *TAAS*, 1(2):223–259, 2006.
[22] S. Dolev. *Self-Stabilization*. MIT Press, 2000.
[23] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997.
[24] N. Elhage and J. Beal. Laplacian-based consensus on spatial computers. In W. van der Hoek, G. A. Kaminka, Y. Lespérance, M. Luck, and S. Sen, editors, *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10-14, 2010, Volume 1-3*, pages 907–914, 2010.
[25] J. L. Fernandez-Marquez, G. D. M. Serugendo, S. Montagna, M. Viroli, and J. L. Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, 12(1):43–67, 2013.
[26] T. Freeman and F. Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.
[27] J.-L. Giavitto, O. Michel, and A. Spicher. Spatial organization of the chemical paradigm and the specification of autonomic systems. volume 5380 of *Lecture Notes in Computer Science*, pages 235–254. Springer, 2008.
[28] M. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering*, 17:911–921, 1991.
[29] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*. Springer, 2004.
[30] T. Herman and I. Pirwani. A composite stabilizing data structure. In *WSS01 Proceedings of the Fifth International Workshop on Self-Stabilizing Systems, Springer LNCS:2194*, pages 167–182, 2001.
[31] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, Oct. 1980.
[32] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.
[33] M. Karaata and P. Chaudhuri. A self-stabilizing algorithm for strong fairness. *Computing*, 60:217–228, 1998.
[34] B. MacLennan. Field computation: A theoretical framework for massively parallel analog computation, parts i-iv. Technical Report Department of Computer Science Technical Report CS-90-100, University of Tennessee, Knoxville, February 1990.
[35] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. on Software Engineering Methodologies*, 18(4):1–56, 2009.
[36] MIT Proto. software available at `http://proto.bbn.com/`, Retrieved January 1st, 2012.
[37] S. Montagna, D. Pianini, and M. Viroli. Gradient-based self-organisation patterns of anticipative adaptation. In *Proceedings of SASO 2012*, pages 169–174. IEEE, September 2012.
[38] S. Montagna, M. Viroli, J. L. Fernandez-Marquez, G. Di Marzo Serugendo, and F. Zambonelli. Injecting self-organisation into pervasive service ecosystems. *Mobile Networks and Applications*, pages 1–15, September 2012. Online first.
[39] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.
[40] A. Omicini and M. Viroli. Coordination models and languages: From parallel computing to self-organisation. *The Knowledge Engineering Review*, 26(1):53–59, Mar. 2011.
[41] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[42] M. Tokoro. Computational field model: toward a new computing model/methodology for open distributed environment. In *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of*, pages 501–506, 1990.
[43] M. Viroli. Engineering confluent computational fields: from functions to rewrite rules. In *Spatial Computing Workshop (SCW 2013)*, AAMAS 2013, May 2013.
[44] M. Viroli, M. Casadei, and A. Omicini. A framework for modelling and implementing self-organising coordination. In *Proceedings of ACM SAC 2009*, volume III, pages 1353–1360. ACM, 8–12 Mar. 2009.
[45] M. Viroli and F. Damiani. A calculus of self-stabilising computational fields. In E. Kühn and R. Pugliese, editors, *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 163–178. Springer Berlin Heidelberg, 2014.
[46] M. Viroli, F. Damiani, and J. Beal. A calculus of computational fields. In C. Canal and M. Villari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg, 2013.
[47] M. Viroli, D. Pianini, and J. Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *Proceedings of Coordination 2012*, volume 7274 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2012.
[48] M. Viroli, D. Pianini, S. Montagna, G. Stevenson, and F. Zambonelli. A coordination model of pervasive service ecosystems. *Science of Computer Programming*, 110:3 – 22, 2015.
[49] D. Weyns, N. Boucké, and T. Holvoet. A field-based versus a protocol-based approach for adaptive task assignment. *Autonomous Agents and Multi-Agent Systems*, 17(2):288–319, 2008.
[50] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004.
[51] F. Zambonelli, G. Castelli, L. Ferrari, M. Mamei, A. Rosi, G. D. M. Serugendo, M. Risoldi, A.-E. Tchao, S. Dobson, G. Stevenson, J. Ye, E. Nardini, A. Omicini, S. Montagna, M. Viroli, A. Ferscha, S. Maschek, and B. Wally. Self-aware pervasive service ecosystems. *Procedia CS*, 7:197–199, 2011.
[52] F. Zambonelli and M. Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 2011.

Appendix A. Proof of Theorem 5.1 and Theorem 5.2

The proof are given for the calculus with pairs (cf. Section 6).

Lemma A.1 (Substitution lemma for typing)**.** *If* $\overline{x} : \overline{T} \vdash e : T$*,* $length(\overline{v}) = length(\overline{x})$ *and* $\emptyset \vdash \overline{v} : \overline{T}$*, then* $\emptyset \vdash e[\overline{x} := \overline{v}] : T$*.*

Proof. Straightforward by induction on the application of the typing rules for expressions in Fig. 4 and Fig. 13. □

Lemma A.2 (Device computation type preservation)**.** *If* $\theta \in WTVT(\overline{x} : \overline{T}, e, T)$*, then* $\emptyset \vdash \rho(\theta) : T$*.*

Proof. Recall that the typing rules (in Fig. 4 and Fig. 13) and the evaluation rules (in Fig. 6 and Fig. 14) are syntax directed. The proof is by induction on the definition of $WTVT(\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{T})$ (given in Section 5.1), on the number of user-defined function calls that may be encountered during the evaluation of $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]$ (cf. sanity condition ***(iii)*** in Section 3.1), and on the syntax of closed expressions.

From the hypothesis $\theta \in WTVT(\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{T})$ we have $\overline{\mathtt{x}} : \overline{\mathtt{T}} \vdash \mathtt{e} : \mathtt{T}$, $\sigma; \overline{\theta} \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] \Downarrow \theta$ for some sensor mapping σ, evaluation trees $\overline{\theta} \in WTVT(\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{T})$, and values $\overline{\mathtt{v}}$ such that $length(\overline{\mathtt{v}}) = length(\overline{\mathtt{x}})$ and $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{T}}$. Moreover, by Lemma A.1 we have that $\emptyset \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] : \mathtt{T}$ holds. The case $\overline{\theta}$ empty represents the base of the induction on the definition of $WTVT(\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{e}, \mathtt{T})$. Therefore the rest of this proof can be understood as a proof of the base step by assuming $\overline{\theta} = \emptyset$ and a proof of the inductive step by assuming $\overline{\theta} \neq \emptyset$.

The case when $\mathtt{e}$ does non contain user-defined function calls represents the base on the induction on the number of user-defined function calls that may be encountered during the evaluation

of $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]$. Therefore the rest of this proof can be understood as a proof of the base step by ignoring the cases $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] = \mathtt{f}(\mathtt{e}_1,...,\mathtt{e}_n)$ and $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] = \{\mathtt{e}_0 : \mathtt{f}(\texttt{@},\mathtt{e}_1,...,\mathtt{e}_n)\}$ when $\mathtt{f}$ is a used-defined function $\mathtt{d}$. The base of the induction on $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]$ consist of two cases.

Case s**:** From the hypothesis we have $\emptyset \vdash s : \mathtt{T}$ where $\mathtt{T} = type(s)$ (by rule [T-SNS]) and $\sigma;\overline{\theta} \vdash s \Downarrow \theta$ where $\theta = \mathtt{v}()$ and $\mathtt{v} = \sigma(s)$ (by rule [E-SNS]). Since the sensor s returns values of type $type(s)$, we have that $type(\mathtt{v}) = type(s) = \mathtt{T}$. So the result follows by a straightforward induction of the syntax of values using rules [T-VAL] and [T-PAIR].

Case $\mathtt{v}$**:** From the hypothesis we have $\emptyset \vdash \mathtt{v} : \mathtt{T}$ and (by rule [E-VAL]) $\sigma;\overline{\theta} \vdash \mathtt{v} \Downarrow \theta$ where $\theta = \mathtt{v}()$. So the result follows by a straightforward induction of the syntax of values using rules [T-VAL] and [T-PAIR].

For the inductive step on $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]$, we show only the two most interesting cases (all the other cases are straightforward by induction).

Case $\mathtt{d}(\mathtt{e}_1,...,\mathtt{e}_n)$**:** From the hypothesis we have $\emptyset \vdash \mathtt{f}(\mathtt{e}_1,...,\mathtt{e}_n) : \mathtt{T}$ (by rule T-FUN]) and $\sigma;\overline{\theta} \vdash \mathtt{d}(\mathtt{e}_1,...,\mathtt{e}_n) \Downarrow \mathtt{v}(\theta'_1,...,\theta'_n,\mathtt{v}(\overline{\eta}))$ (by rule E-DEF]). Therefore we have $\mathtt{T}(\mathtt{T}_1,...,\mathtt{T}_n) = t\text{-}sig(\mathtt{d})$, $\emptyset \vdash \mathtt{e}_1 : \mathtt{T}_1$, ..., $\emptyset \vdash \mathtt{e}_n : \mathtt{T}_n$ (the premises of rule T-FUN]) and $\mathtt{def\ T\ d}(\mathtt{T}_1\ \mathtt{x}_1,...,\mathtt{T}_n\ \mathtt{x}_n) = \mathtt{e}''$, $\sigma;\pi_1(\overline{\theta}) \vdash \mathtt{e}_1 \Downarrow \theta'_1$, ..., $\sigma;\pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \theta'_n$ and

$$\sigma;\pi_{n+1}(\overline{\theta}) \vdash \mathtt{e}' \Downarrow \mathtt{v}(\overline{\eta}) \quad \text{where } \mathtt{e}' = \mathtt{e}''[\mathtt{x}_1 := \rho(\theta'_1),...,\mathtt{x}_n := \rho(\theta'_n)] \tag{A.1}$$

(the premises of rule E-DEF]).

Since $\pi_i(\overline{\theta}) \in WTVT(\emptyset,\mathtt{e}_i,\mathtt{T}_i)$ $(1 \le i \le n)$ then $\theta'_i \in WTVT(\emptyset,\mathtt{e}_i,\mathtt{T}_i)$; therefore, by induction we have $\emptyset \vdash \rho(\theta'_1) : \mathtt{T}_1$, ..., $\emptyset \vdash \rho(\theta'_n) : \mathtt{T}_n$.

Since the program is well typed (cf. Section 3.1) we have $\mathtt{x}_1 : \mathtt{T}_1,...,\mathtt{x}_n : \mathtt{T}_n \vdash \mathtt{e}'' : \mathtt{T}$ (by rule T-DEF]).

Since $\pi_{n+1}(\overline{\theta}) \in WTVT(\mathtt{x}_1 : \mathtt{T}_1,...,\mathtt{x}_n : \mathtt{T}_n,\mathtt{e}'',\mathtt{T})$, then (by (A.1)) we have $\mathtt{v}(\overline{\eta}) \in WTVT(\mathtt{x}_1 : \mathtt{T}_1,...,\mathtt{x}_n : \mathtt{T}_n,\mathtt{e}'',\mathtt{T})$; therefore, by induction we have that $\emptyset \vdash \mathtt{v} : \mathtt{T}$.

Case $\{\mathtt{e}_0 : \mathtt{f}(\texttt{@},\mathtt{e}_1,...,\mathtt{e}_n)\}$**:** From the hypothesis we have $\emptyset \vdash \{\mathtt{e}_0 : \mathtt{f}(\texttt{@},\mathtt{e}_1,...,\mathtt{e}_n)\} : \mathtt{T}$ (by rule T-SPR]) and $\sigma;\overline{\theta} \vdash \{\mathtt{e}_0 : \mathtt{f}(\texttt{@},\mathtt{e}_1,...,\mathtt{e}_n)\} \Downarrow \bigwedge\{\mathtt{v}_0,\mathtt{u}_1,...,\mathtt{u}_m\}(\eta_0,\eta_1,...,\eta_n)$ (by rule E-SPR]). Therefore we have $diffusion(\mathtt{f})$, $\mathtt{T}(\mathtt{T},\mathtt{T}_1,...,\mathtt{T}_n) = t\text{-}sig(\mathtt{d})$, $\emptyset \vdash \mathtt{e}_0 : \mathtt{T}$, $\emptyset \vdash \mathtt{e}_1 : \mathtt{T}_1$, ..., $\emptyset \vdash \mathtt{e}_n : \mathtt{T}_n$ (the premises of rules T-SPR] and T-FUN]) and $\sigma;\pi_1(\overline{\theta}) \vdash \mathtt{e}_1 \Downarrow \theta'_1$, ..., $\sigma;\pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \theta'_n$, $\rho(\eta_0,...,\eta_n) = \mathtt{v}_0...\mathtt{v}_n$, $\rho(\overline{\theta}) = \mathtt{w}_1...\mathtt{w}_m$,

$$\sigma;\emptyset \vdash \mathtt{f}(\mathtt{w}_1,\mathtt{v}_1,...,\mathtt{v}_n) \Downarrow \mathtt{u}_1(\cdots),...,\sigma;\emptyset \vdash \mathtt{f}(\mathtt{w}_m,\mathtt{v}_1,...,\mathtt{v}_n) \Downarrow \mathtt{u}_m(\cdots) \tag{A.2}$$

(the premises rule E-SPR]). By induction we have $\emptyset \vdash \mathtt{v}_0...\mathtt{v}_n : \mathtt{T}\mathtt{T}_1...\mathtt{T}_n$ and $\emptyset \vdash \mathtt{w}_1 : \mathtt{T}$, ..., $\emptyset \vdash \mathtt{w}_m : \mathtt{T}$. We have two subcases:

- If $\mathtt{f}$ is a user-defined function, then from (A.2) we get (by reasoning as in the proof of case $\mathtt{d}(\mathtt{e}_1,...,\mathtt{e}_n)$) $\emptyset \vdash \mathtt{u}_1 : \mathtt{T}$, ..., $\emptyset \vdash \mathtt{u}_m : \mathtt{T}$.
- If $\mathtt{f}$ is a built-in function, then from (A.2) we get (by the semantics of built-in functions) $\emptyset \vdash \mathtt{u}_1 : \mathtt{T}$, ..., $\emptyset \vdash \mathtt{u}_m : \mathtt{T}$.

In both cases $\mathtt{v} = \bigwedge\{\mathtt{v}_0,\mathtt{u}_1,...,\mathtt{u}_m\}$ has type $\mathtt{T}$, i.e., $\emptyset \vdash \mathtt{v} : \mathtt{T}$ holds. □

Restatement of Theorem 5.1 (Device computation type preservation). *If* $\overline{x} : \overline{T} \vdash e : T$, σ *is a sensor mapping,* $\overline{\theta} \in WTVT(\overline{x} : \overline{T}, e, T)$, $length(\overline{v}) = length(\overline{x})$, $\emptyset \vdash \overline{v} : \overline{T}$ *and* $\sigma;\overline{\theta} \vdash e[\overline{x} := \overline{v}] \Downarrow \theta$, *then* $\emptyset \vdash \rho(\theta) : T$.

Proof. Straightforward by Lemma A.2, since $\theta \in WTVT(\overline{\mathtt{x}} : \overline{\mathtt{T}},\mathtt{e},\mathtt{T})$. □

Restatement of Theorem 5.2 (Device computation termination). *If* $\overline{x} : \overline{T} \vdash e : T$*,* σ *is a sensor mapping,* $\overline{\theta} \in WTVT(\overline{x} : \overline{T}, e, T)$*,* $length(\overline{v}) = length(\overline{x})$ *and* $\emptyset \vdash \overline{v} : \overline{T}$*, then* $\sigma; \overline{\theta} \vdash e[\overline{x} := \overline{v}] \Downarrow \theta$ *for some value-tree* θ*.*

Proof. By induction on the number of function calls that may be encountered during the evaluation of $e[\overline{x} := \overline{v}]$ (cf. sanity condition ***(iii)*** in Section 3.1) and on the syntax of closed expressions, using Lemma A.2. □

Appendix B. Proof of Theorem 5.3

Recall the auxiliary definitions and the outline of the proof of Theorem 5.3 given in Section 5.2. The following six lemmas corresponds to the auxiliary results B.1-B.6 introduced in Section 5.2.

Lemma B.1 (Minimum value)**.** *Given a program* $e = \{e_0 : f(@, e_1, \ldots, e_n)\}$ *with valid sort and stabilising diffusion assumptions, for every reachable pre-self-stable network configuration N, for any device* ι *in N such that* $v_{\iota(in\,N)}$ *is minimum (among the values of the devices in N):*

(1) *if* $N \overset{\overline{\iota}}{\Longrightarrow} N'$ *and* $\iota \notin \overline{\iota}$*, then* $v_{\iota(in\,N')} = v_{\iota(in\,N)}$ *is minimum (among the values of the devices in* N'*); and*
(2) *either* $v_{\iota(in\,N)} = v_{0,\iota}$ *(i.e.,* ι *is self-stable in N) or if* $N \Longrightarrow_1 N'$ *then there exists* v' *such that:*
   (a) $v_{\iota(in\,N)} < v' \le v_{\iota(in\,N')}$*; and*
   (b) *if* $N' \Longrightarrow N''$*, then* $v' \le v_{\iota(in\,N'')}$*.*

Proof. **Point (1).** Consider a device $\iota' \neq \iota$. Then $v_{\iota(\text{in}\,N)} \le v_{\iota'(\text{in}\,N)} \le v_{0,\iota'}$ and its $m \ge 0$ neighbours have values w_j such that $v_{\iota(\text{in}\,N)} \le w_j$ $(1 \le j \le m)$. Since the stabilising diffusion assumptions hold, $w_j < [\![f]\!](w_j, v_{1,\iota'}, \ldots, v_{n,\iota'}) = u_j$. Therefore, $v_{\iota(\text{in}\,N)} \le \bigwedge\{v_{0,\iota'}, u_1, ..., u_m\} = v_{\iota'(\text{in}\,N')}$. So $v_{\iota(\text{in}\,N')} = v_{\iota(\text{in}\,N)}$ is minimum (among the values of the devices in N').
Point (2). Assume that $v_{\iota(\text{in}\,N)} < v_{0,\iota}$. The $m \ge 0$ neighbours of ι have values w_j such that $v_{\iota(\text{in}\,N)} \le w_j$ $(1 \le j \le m)$. Since the stabilising diffusion assumptions hold, $v_{\iota(\text{in}\,N)} < [\![f]\!](v_{\iota(\text{in}\,N)}, v_{1,\iota}, \ldots, v_{n,\iota}) = u_0$ and $u_0 \le [\![f]\!](w_j, v_{1,\iota}, \ldots, v_{n,\iota}) = u_j$. Therefore, the value $v' = v_{0,\iota} \wedge u_0$ is such that when ι fires its new value $v_{\iota(\text{in}\,N')} = \bigwedge\{v_{0,\iota}, u_1, ..., u_m\}$ is such that $v_{\iota(\text{in}\,N)} < v' \le v_{\iota(\text{in}\,N')}$. Moreover, since $v_{\iota(\text{in}\,N)} < v_{\iota(\text{in}\,N')}$ we have that in a firing evolution none of devices $\iota' \neq \iota$ will reach a value less than $v_{\iota(\text{in}\,N)}$ and therefore the device ι will never reach a value less than v'. □

Lemma B.2 (Self-stabilisation of the minimum value)**.** *Given a program* $e = \{e_0 : f(@, e_1, \ldots, e_n)\}$ *with valid sort and stabilising diffusion assumptions, for every reachable pre-self-stable network configuration N, if* $\boldsymbol{S}_1$ *is the subset of the devices in N such that* $v_{0,\iota}$ *is minimum (among the values of* e_0 *in the devices in N), then there exists* $k \ge 0$ *such that* $N \Longrightarrow_k N'$ *implies that* $\boldsymbol{S}_1$ *satisfies the following conditions:*

(1) *each device* ι *in* $\boldsymbol{S}_1$ *is self-stable in* N' *and has value* $v_{\iota(in\,N')} = v_{0,\iota}$*;*
(2) *in* N' *each device not in* $\boldsymbol{S}_1$ *has a value greater or equal to the values of the devices in* $\boldsymbol{S}_1$ *and, during any firing evolution, it will always assume values greater than the values of the devices in* $\boldsymbol{S}_1$*.*

Proof. The number of devices in the network configuration is finite, the environment does not change, the network is pre-self-stable, and the stabilising diffusion assumptions holds. The results follows by Lemma B.1. Namely, if there is a device ι whose value v_ι is minimum and such that $v_\iota < v_{0,\iota}$, then after a 1-fair network evolution ι reaches a value which is greater or equal to some v' such that

- $v < v' \le v_{0,\iota}$; and

- in any subsequent firing evolution the value of ι will be always greater or equal to $\mathtt{v}'$.

Therefore, after a finite number k of 1-fair evolutions (i.e., after any k-fair evolution) conditions (1) and (2) in the statement of the lemma are satisfied. □

Lemma B.3 (Frontier)**.** *Given a program* $\mathsf{e} = \{\mathsf{e}_0 : \mathsf{f}(@, \mathsf{e}_1, \ldots, \mathsf{e}_n)\}$ *with valid sort and stabilising diffusion assumptions, for every reachable pre-stable network configuration* N *with devices* $\boldsymbol{D}$ *and a non-empty subset of devices* $\boldsymbol{S} \subset \boldsymbol{D}$ *such that*

(i) *each device in* $\boldsymbol{S}$ *is self-stable in* N*;*
(ii) *each device in* $\boldsymbol{D}-\boldsymbol{S}$ *has a value greater or equal to the values of the devices in* $\boldsymbol{S}$ *and, during any firing evolution, will always assume values greater or equal to the values of the devices in* $\boldsymbol{S}$*; and*
(iii) $\mathit{frontier}_{\boldsymbol{S}}(\boldsymbol{D}) \neq \emptyset$*;*

if $N \Longrightarrow_1 N'$ *then each device in* $\mathit{frontier}_{\boldsymbol{S}}(\boldsymbol{D})$ *is self-stable in* N'*.*

Proof. When a device ι in $\mathit{frontier}_{\mathbf{S}}(\mathbf{D})$ fires it gets the value

$$\mathtt{v}_{0,\iota} \wedge [\![\mathtt{f}]\!](\mathtt{v}_{\iota'}, \mathtt{v}_{1,\iota}, \ldots, \mathtt{v}_{n,\iota})$$

where $\iota' \in \mathbf{S}$ is the neighbour of ι that has minimum value (among the neighbours of ι). This value is univocally determined by *environment*(N) and is stable, since the values $\mathtt{v}_{\iota'}$ and $\mathtt{v}_{j,\iota}$ $(0 \le j \le n)$ are stable and in any firing evolution each neighbour of ι assumes only values greater or equal to $\mathtt{v}_{\iota'}$. □

Note that conditions (i)-(iii) of the following lemma are exactly the same as in Lemma B.3.

Lemma B.4 (Minimum value not in **S**)**.** *Given a program* $\mathsf{e} = \{\mathsf{e}_0 : \mathsf{f}(@, \mathsf{e}_1, \ldots, \mathsf{e}_n)\}$ *with valid sort and stabilising diffusion assumptions, for every reachable pre-self-stable network configuration* N *with devices* $\boldsymbol{D}$ *and a non-empty subset of devices* $\boldsymbol{S} \subset \boldsymbol{D}$ *such that*

(i) *each device in* $\boldsymbol{S}$ *is self-stable in* N*;*
(ii) *each device in* $\boldsymbol{D}-\boldsymbol{S}$ *has a value greater or equal to the values of the devices in* $\boldsymbol{S}$ *and, during any firing evolution, will always assume values greater or equal to the values of the devices in* $\boldsymbol{S}$*;*
(iii) $\mathit{frontier}_{\boldsymbol{S}}(\boldsymbol{D}) \neq \emptyset$*; and*
(iv) *each device in* $\mathit{frontier}_{\boldsymbol{S}}(\boldsymbol{D})$ *is self-stable in* N*;*

if $\boldsymbol{M} \subseteq \boldsymbol{D}-\boldsymbol{S}$ *is the set of devices* ι *such that* $v_{\iota(\mathit{in}\,N)}$ *is minimum (among the values of the devices in* $\boldsymbol{D}-\boldsymbol{S}$*), then*

(1) *if* $\iota \in \boldsymbol{M}$*,* $N \stackrel{\bar{\iota}}{\Longrightarrow} N'$ *and* $\iota \notin \bar{\iota}$*, then* $v_{\iota(\mathit{in}\,N)} = v_{\iota(\mathit{in}\,N')}$ *is minimum (among the values of the devices in* $\boldsymbol{D}-\boldsymbol{S}$ *in* N'*);*
(2) *if* $\boldsymbol{M} \cap \mathit{frontier}_{\boldsymbol{S}}(\boldsymbol{D}) = \emptyset$*, then there is a device* $\iota \in \boldsymbol{M}$ *such that either* $v_{\iota(\mathit{in}\,N)} = v_{0,\iota}$ *(i.e.,* ι *is self-stable in* N*) or if* $N \Longrightarrow_1 N'$ *then there exists* v' *such that:*
    (a) $v_{\iota(\mathit{in}\,N)} < v' \le v_{\iota(\mathit{in}\,N')}$*; and*
    (b) *if* $N' \Longrightarrow N''$*, then* $v' \le v_{\iota(\mathit{in}\,N'')}$*.*

Proof. Since the self-stable values in $\mathit{frontier}_{\mathbf{S}}(\mathbf{D})$ ensure that in any firing evolution the values of the devices in $\mathbf{D}-(\mathbf{S}\cup\mathit{frontier}_{\mathbf{S}}(\mathbf{D}))$ are computed without using the values of the devices in $\mathbf{S}$, the proof is similar to the proof of Lemma B.1.

Point (1). Consider a device $\iota' \notin \mathbf{S}\cup\mathbf{M}$. Then $\mathtt{v}_{\iota(\mathrm{in}\,N)} \le \mathtt{v}_{\iota'(\mathrm{in}\,N)} \le \mathtt{v}_{0,\iota'}$ and its $m \ge 0$ neighbours have values $\mathtt{w}_j$ such that $\mathtt{v}_{\iota(\mathrm{in}\,N)} \le \mathtt{w}_j$ $(1 \le j \le m)$. Since the stabilising diffusion assumptions hold, $\mathtt{w}_j < [\![\mathtt{f}]\!](\mathtt{w}_j, \mathtt{v}_{1,\iota'}, \ldots, \mathtt{v}_{n,\iota'}) = \mathtt{u}_j$. Therefore, $\mathtt{v}_{\iota(\mathrm{in}\,N)} \le \bigwedge\{\mathtt{v}_{0,\iota'}, \mathtt{u}_1, ..., \mathtt{u}_m\} = \mathtt{v}_{\iota'(\mathrm{in}\,N')}$. So $\mathtt{v}_{\iota(\mathrm{in}\,N')} = \mathtt{v}_{\iota(\mathrm{in}\,N)}$ is minimum (among the values of the devices of $\mathbf{D}-\mathbf{S}$ in N').

Point (2). Assume that $\mathtt{v}_{\iota(\text{in }N)} < \mathtt{v}_{0,\iota}$. The $m \geq 0$ neighbours of ι have values $\mathtt{w}_j$ such that $\mathtt{v}_{\iota(\text{in }N)} \leq \mathtt{w}_j$ ($1 \leq j \leq m$). Since the stabilising diffusion assumptions hold, $\mathtt{v}_{\iota(\text{in }N)} < [\![\mathtt{f}]\!](\mathtt{v}_{\iota(\text{in }N)}, \mathtt{v}_{1,\iota}, \ldots, \mathtt{v}_{n,\iota}) = \mathtt{u}_0$ and $\mathtt{u}_0 \leq [\![\mathtt{f}]\!](\mathtt{w}_j, \mathtt{v}_{1,\iota}, \ldots, \mathtt{v}_{n,\iota}) = \mathtt{u}_j$. Therefore, the value $\mathtt{v}' = \mathtt{v}_{0,\iota} \wedge \mathtt{u}_0$ is such that when ι fires its new value $\mathtt{v}_{\iota(\text{in }N')} = \bigwedge\{\mathtt{v}_{0,\iota}, \mathtt{u}_1, ..., \mathtt{u}_m\}$ is such that $\mathtt{v}_{\iota(\text{in }N)} < \mathtt{v}' \leq \mathtt{v}_{\iota(\text{in }N')}$. Moreover, since $\mathtt{v}_{\iota(\text{in }N)} < \mathtt{v}_{\iota(\text{in }N')}$ we have that in a firing evolution none of the devices $\iota' \in \mathbf{D} - (\mathbf{S} \cup \{\iota\})$ will reach a value less than $\mathtt{v}_{\iota(\text{in }N)}$ and therefore the device ι will never reach a value less than $\mathtt{v}'$. □

Note that conditions *(i)-(iv)* of the following lemma are exactly the same as in Lemma B.4.

Lemma B.5 (Self-stabilisation of the minimum value not in **S**)**.** *Given a program* $\mathsf{e} = \{\mathsf{e}_0 : f(@, \mathsf{e}_1, \ldots, \mathsf{e}_n)\}$ *with valid sort and stabilising diffusion assumptions, for every reachable pre-self-stable network configuration* N *with devices* $\boldsymbol{D}$ *and a non-empty subset of devices* $\boldsymbol{S} \subset \boldsymbol{D}$ *such that*

(i) *each device in* $\boldsymbol{S}$ *is self-stable in* N*;*
(ii) *each device in* $\boldsymbol{D} - \boldsymbol{S}$ *has a value greater or equal to the values of the devices in* $\boldsymbol{S}$ *and, during any firing evolution, will always assume values greater or equal to the values of the devices in* $\boldsymbol{S}$*;*
(iii) $\mathit{frontier}_{\boldsymbol{S}}(\boldsymbol{D}) \neq \emptyset$*; and*
(iv) *each device in* $\mathit{frontier}_{\boldsymbol{S}}(\boldsymbol{D})$ *is self-stable in* N*;*

there exists $k \geq 0$ *such that* $N \Longrightarrow_k N'$ *implies that there exists a device* ι_1 *in* $\boldsymbol{D} - \boldsymbol{S}$ *such that* $\boldsymbol{S}_1 = \boldsymbol{S} \cup \{\iota_1\}$ *satisfies the following conditions:*

(1) *each device* ι *in* $\boldsymbol{S}_1$ *is self-stable in* N'*; and*
(2) *in* N' *any device in* $\boldsymbol{D} - \boldsymbol{S}_1$ *has a value greater or equal to the values of the devices in* $\boldsymbol{S}_1$ *and, during any firing evolution, will always assume values greater than the values of the devices in* $\boldsymbol{S}_1$.

Proof. Let $\mathbf{M} \subseteq \mathbf{D} - \mathbf{S}$ be the set of devices ι such that $\mathtt{v}_{\iota(\text{in }N)}$ is minimum (among the values of the devices in $\mathbf{D} - \mathbf{S}$). We consider two cases.

$\mathbf{M} \cap \boldsymbol{frontier}_{\mathbf{S}}(\mathbf{D}) \neq \emptyset$**:** Any of the devices $\iota_1 \in \mathbf{M} \cap \mathit{frontier}_{\mathbf{S}}(\mathbf{D})$ is such that conditions (1) and (2) in the statement of the lemma are satisfied.

$\mathbf{M} \cap \boldsymbol{frontier}_{\mathbf{S}}(\mathbf{D}) = \emptyset$**:** The number of devices in the network configuration is finite, the environment does not change, the network is pre-self-stable, and the stabilising diffusion condition holds. The results follows by Lemma B.4.[9] Namely, if there is a device $\iota \in \mathbf{M}$ such that $\mathtt{v}_\iota < \mathtt{v}_{0,\iota}$, then after a 1-fair network evolution ι reaches a value which is greater or equal to some $\mathtt{v}'$ such that
- $\mathtt{v} < \mathtt{v}' \leq \mathtt{v}_{0,\iota}$; and
- in any subsequent firing evolution the value of ι will be always greater or equal to $\mathtt{v}'$.

Therefore, after a finite number k of 1-fair evolutions (i.e., after any h-fair evolution with $h \geq k$) conditions (1) and (2) in the statement of the lemma are satisfied. □

Lemma B.6 (Pre-self-stable network self-stabilization)**.** *Given a program* $\mathsf{e} = \{\mathsf{e}_0 : f(@, \mathsf{e}_1, \ldots, \mathsf{e}_n)\}$ *with valid sort and stabilising diffusion assumptions, for every reachable pre-self-stable network configuration* N *there exists* $k \geq 0$ *such that* $N \Longrightarrow_k N'$ *implies that* N' *is self-stable.*

Proof. Let $\mathbf{D}$ be a set of devices of N. The proof is by induction on the number of devices in $\mathbf{D}$.

Case $\mathbf{D} = \emptyset$**:** Immediate.

[9] In particular, since the self-stable values in $\mathit{frontier}_{\mathbf{S}}(\mathbf{D})$ ensure that in any firing evolution the values of the devices in $\mathbf{D} - (\mathbf{S} \cup \mathit{frontier}_{\mathbf{S}}(\mathbf{D}))$ are computed without using the values of the devices in $\mathbf{S}$, the proof of this case is similar to the proof of Lemma B.2 (by using Lemma B.4 instead of Lemma B.1).

Case $\mathbf{D} \neq \emptyset$: By Lemma B.2 there exists $k_0 \geq 0$ such that after any k_0-fair evolution there is a non-empty set of devices $\mathbf{S}_1$ that satisfies conditions (1) and (2) in the statement of Lemma B.2 (and, therefore, also conditions (1) and (2) in the statement of Lemma B.5).

Now rename $\mathbf{S}_1$ to $\mathbf{S}$, consider a counter c intially equal to 1, and iterate the following two reasoning steps while the set of devices $\mathbf{S}$ is such that $\mathit{frontier}_{\mathbf{S}}(\mathbf{D}) \neq \emptyset$:

- By lemma B.3 and lemma B.5 there exists $k_c \geq 0$ such that after any k_c-fair evolution there is a non-empty set of devices $\mathbf{S}_1$ that satisfies conditions (1) and (2) in the statement of Lemma B.5.
- Rename $\mathbf{S}_1$ to $\mathbf{S}$ and increment the value of c.

Since at each iteration the number devices in $\mathbf{S}$ is increased by one, the number of iterations is finite (note that $\mathbf{S} = \mathbf{D}$ implies $\mathit{frontier}_{\mathbf{S}}(\mathbf{D}) = \emptyset$). After the last iteration, we have proved that there exists $k' = \Sigma_{j=0}^{c-1} k_j$ such that after any k'-fair evolution there is a non-empty set of devices $\mathbf{S}$ that satisfies conditions (1) and (2) in the statement of Lemma B.5. Since $\mathit{frontier}_{\mathbf{S}}(\mathbf{D}) = \emptyset$, then the evolution of the devices in $\mathbf{D} - \mathbf{S}$ is independent from the devices in $\mathbf{S}$. By induction there exists $k'' \geq 0$ such that after any k''-fair evolution the portion of the network with devices in $\mathbf{D} - \mathbf{S}$ is self-stable. Therefore, we have proved the lemma with $k = k' + k''$. □

Restatement of Theorem 5.3 (Network self-stabilisation for programs that satisfy the stabilising-diffusion condition)**.** *Given a program with valid sort and stabilising diffusion assumptions, every reachable network configuration N self-stabilises, i.e., there exists $k \geq 0$ such that $N \Longrightarrow_k N'$ implies that N' is self-stable.*

Proof. By induction on the syntax of closed expressions $\mathtt{e}$ and on the number of function calls that may be encountered during the evaluation of $\mathtt{e}$. Let $E = \mathit{environment}(N)$.

Case $\mathtt{v}$: Any device fire produces the value-tree $\mathtt{v}()$ (independently from E).

Case s: Each fire of device ι produces the value-tree $\mathtt{v}()$, where $\mathtt{v} = \sigma_\iota(s)$ is univocally determined by E.

Case $\mathtt{b}(\overline{\mathtt{e}})$: Straightforward by induction.

Case $\mathtt{f}(\overline{\mathtt{e}})$: Straightforward by induction.

Case $\{\mathtt{e}_0 : \mathtt{f}(@, \mathtt{e}_1, \ldots, \mathtt{e}_n)\}$: By induction there exists $h \geq 0$ such that if $N \Longrightarrow_h N_1$ then on every device ι, the evaluation of $\mathtt{e}_0, \mathtt{e}_1, \ldots, \mathtt{e}_n$ produce stable value-trees $\theta_{0,\iota}, \theta_{1,\iota}, \ldots, \theta_{n,\iota}$, which are univocally determined by E. Note that, if $N \Longrightarrow_{h+1} N_2$ then N_2 is pre-self-stable. Therefore the result follows straightforwardly by Lemma B.6. □

Appendix C. Proof of Theorem 8.6

The proof of Theorem 8.6 is similar to the proof of Theorem 5.1 (cf. Appendix A)—see Remark 8.7.

Lemma C.1 (Substitution lemma for sorting)**.** *If $\overline{\mathtt{x}} : \overline{S} \vdash \mathtt{e} : S$, $\mathit{length}(\overline{\mathtt{v}}) = \mathit{length}(\overline{\mathtt{x}})$, $\emptyset \vdash \overline{\mathtt{v}} : \overline{S'}$, and $\overline{S'} \leq \overline{S}$, then $\emptyset \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] : S'$ for some S' such that $S' \leq S$.*

Proof. Straightforward by induction on the application of the sort-checking rules for expressions in Fig. 19. □

Lemma C.2 (Device computation sort preservation)**.** *If $\theta \in WSVT(\overline{x} : \overline{S}, e, S)$, then $\emptyset \vdash \rho(\theta) : S'$ for some S' such that $S' \leq S$.*

Proof. Recall that the sorting rules (in Fig. 19) and the evaluation rules (in Fig. 6 and Fig. 14) are syntax directed. The proof is by induction on the definition of $WSVT(\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{e}, \mathtt{S})$ (given in Section 8.6), on the number of user-defined function calls that may be encountered during the evaluation of $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]$ (cf. sanity condition ***(iii)*** in Section 3.1), and on the syntax of closed expressions.

From the hypothesis $\theta \in WSVT(\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{e}, \mathtt{S})$ we have $\overline{\mathtt{x}} : \overline{\mathtt{S}} \vdash \mathtt{e} : \mathtt{S}$, $\sigma; \overline{\theta} \vdash \mathtt{e} \Downarrow \theta$ for some sensor mapping σ, evaluation trees $\overline{\theta} \in WSVT(\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{e}, \mathtt{S})$, and values $\overline{\mathtt{v}}$ such that $length(\overline{\mathtt{v}}) = length(\overline{\mathtt{x}})$, $\emptyset \vdash \overline{\mathtt{v}} : \overline{\mathtt{S}}'$, and $\overline{\mathtt{S}}' \leq \overline{\mathtt{S}}$. The case $\overline{\theta}$ empty represents the base of the induction on the definition of $WSVT(\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{e}, \mathtt{S})$. Therefore the rest of this proof can be understood as a proof of the base step by assuming $\overline{\theta} = \emptyset$ and a proof of the inductive step by assuming $\overline{\theta} \neq \emptyset$. Moreover, by Lemma C.1 we have that $\emptyset \vdash \mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] : \mathtt{S}'$ for some $\mathtt{S}'$ such that $\mathtt{S}' \leq \mathtt{S}$.

The case when $\mathtt{e}$ does non contain user-defined function calls represents the base on the induction on the number of user-defined function calls that may be encountered during the evaluation of $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]$. Therefore the rest of this proof can be understood as a proof of the base step by ignoring the cases $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] = \mathtt{f}(\mathtt{e}_1, ..., \mathtt{e}_n)$ and $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}] = \{\mathtt{e}_0 : \mathtt{f}(@, \mathtt{e}_1, ..., \mathtt{e}_n)\}$ when $\mathtt{f}$ is a used-defined function $\mathtt{d}$. The base of the induction on $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]$ consist of two cases.

Case s**:** From the hypothesis we have $\emptyset \vdash s : \mathtt{S}$ where $\mathtt{S} = sort(s)$ (by rule [T-SNS]) and $\sigma; \overline{\theta} \vdash s \Downarrow \theta$ where $\theta = \mathtt{v}()$ and $\mathtt{v} = \sigma(s)$ (by rule [E-SNS]). Since the sensor s returns values of sort $sort(s)$, we have that $sort(\mathtt{v}) \leq sort(s) = \mathtt{S}$. So the result follows by a straightforward induction of the syntax of values using rules [S-VAL] and [S-PAIR].

Case $\mathtt{v}$**:** From the hypothesis we have $\emptyset \vdash \mathtt{v} : \mathtt{S}$ and (by rule [E-VAL]) $\sigma; \overline{\theta} \vdash \mathtt{v} \Downarrow \theta$ where $\theta = \mathtt{v}()$. So the result follows by a straightforward induction of the syntax of values using rules [S-VAL] and [S-PAIR].

For the inductive step on $\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]$, we show only the two most interesting cases (all the other cases are straightforward by induction).

Case $\mathtt{d}(\mathtt{e}_1, ..., \mathtt{e}_n)$**:** From the hypothesis we have $\emptyset \vdash \mathtt{f}(\mathtt{e}_1, ..., \mathtt{e}_n) : \mathtt{S}$ (by rule S-FUN]) and $\sigma; \overline{\theta} \vdash \mathtt{d}(\mathtt{e}_1, ..., \mathtt{e}_n) \Downarrow \mathtt{v}(\theta'_1, ..., \theta'_n, \mathtt{v}(\overline{\eta}))$ (by rule E-DEF]). Therefore we have $\mathtt{S}(\mathtt{S}_1, ..., \mathtt{S}_n) = ms(s\text{-}sigs(\mathtt{f}), \mathtt{S}'_1, \ldots, \mathtt{S}'_n)$, $\emptyset \vdash \mathtt{e}_1 : \mathtt{S}'_1$, ..., $\emptyset \vdash \mathtt{e}_n : \mathtt{S}'_n$ and $\mathtt{S}'_1 \leq \mathtt{S}_1$, ..., $\mathtt{S}'_n \leq \mathtt{S}_n$ (by the premises of rule S-FUN]) and $\mathtt{def\ T\ d}(\mathtt{T}_1\ \mathtt{x}_1, ..., \mathtt{T}_n\ \mathtt{x}_n) = \mathtt{e}''$, $\sigma; \pi_1(\overline{\theta}) \vdash \mathtt{e}_1 \Downarrow \theta'_1$, ..., $\sigma; \pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \theta'_n$ and

$$\sigma; \pi_{n+1}(\overline{\theta}) \vdash \mathtt{e}' \Downarrow \mathtt{v}(\overline{\eta}) \quad \text{where } \mathtt{e}' = \mathtt{e}''[\mathtt{x}_1 := \rho(\theta'_1), ..., \mathtt{x}_n := \rho(\theta'_n)] \tag{C.1}$$

(the premises of rule E-DEF]).

Since $\pi_i(\overline{\theta}) \in WSVT(\emptyset, \mathtt{e}_i, \mathtt{S}'_i)$ $(1 \leq i \leq n)$, then $\theta'_i \in WSVT(\emptyset, \mathtt{e}_i, \mathtt{S}'_i)$; therefore, by induction we have $\emptyset \vdash \rho(\theta'_1) : \mathtt{S}''_1$, ..., $\emptyset \vdash \rho(\theta''_n) : \mathtt{S}''_n$ and $\mathtt{S}''_1 \leq \mathtt{S}'_1 \leq \mathtt{S}_1$, ..., $\mathtt{S}''_n \leq \mathtt{S}'_n \leq \mathtt{S}_n$.

Since the program is well sorted (cf. Section 8.5) we have $\mathtt{x}_1 : \mathtt{S}_1, ..., \mathtt{x}_n : \mathtt{S}_n \vdash \mathtt{e}'' : \mathtt{S}'$ and $\mathtt{S}' \leq \mathtt{S}$ (by rule T-DEF]).

Since $\pi_{n+1}(\overline{\theta}) \in WTVT(\mathtt{x}_1 : \mathtt{S}_1, ..., \mathtt{x}_n : \mathtt{S}_n, \mathtt{e}'', \mathtt{S}')$, then (by (C.1)) we have $\mathtt{v}(\overline{\eta}) \in WTVT(\mathtt{x}_1 : \mathtt{S}_1, ..., \mathtt{x}_n : \mathtt{S}_n, \mathtt{e}'', \mathtt{S}')$; therefore, by induction we have that $\emptyset \vdash \mathtt{v} : \mathtt{S}''$ with $\mathtt{S}'' \leq \mathtt{S}' \leq \mathtt{S}$.

Case $\{\mathtt{e}_0 : \mathtt{f}(@, \mathtt{e}_1, ..., \mathtt{e}_n)\}$**:** From the hypothesis we have $\emptyset \vdash \{\mathtt{e}_0 : \mathtt{f}(@, \mathtt{e}_1, ..., \mathtt{e}_n)\} : \mathtt{S}'$ (by rule S-SPR]) and $\sigma; \overline{\theta} \vdash \{\mathtt{e}_0 : \mathtt{f}(@, \overline{\mathtt{e}})\} \Downarrow \bigwedge\{\mathtt{v}_0, \mathtt{u}_1, ..., \mathtt{u}_m\}(\eta_0, \eta_1, ..., \eta_n)$ (by rule E-SPR]). Therefore we have $diffusion(\mathtt{f})$, $\mathscr{S} \vdash \mathtt{e}_0\overline{\mathtt{e}} : \mathtt{S}'_0\overline{\mathtt{S}}'$, $\mathtt{S}'(\mathtt{S}_0\overline{\mathtt{S}}) = ms(stb\text{-}s\text{-}sigs(\mathtt{f}), \mathtt{S}'_0\overline{\mathtt{S}}')$, $\mathtt{S} = sup_{\leq}(\mathtt{S}'_0, \mathtt{S}')$ and $\mathtt{S}'_0\overline{\mathtt{S}}' \leq \mathtt{S}_0\overline{\mathtt{S}}$ (by the premises of rule S-SPR]) and $\sigma; \pi_1(\overline{\theta}) \vdash \mathtt{e}_1 \Downarrow \theta'_1$, ..., $\sigma; \pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \theta'_n$, $\rho(\eta_0, ..., \eta_n) = \mathtt{v}_0 ... \mathtt{v}_n$, $\rho(\overline{\theta}) = \mathtt{w}_1 ... \mathtt{w}_m$,

$$\sigma; \emptyset \vdash \mathtt{f}(\mathtt{w}_1, \mathtt{v}_1, ..., \mathtt{v}_n) \Downarrow \mathtt{u}_1(\cdots), ..., \sigma; \emptyset \vdash \mathtt{f}(\mathtt{w}_m, \mathtt{v}_1, ..., \mathtt{v}_n) \Downarrow \mathtt{u}_m(\cdots) \tag{C.2}$$

(the premises rule E-SPR]). By induction we have $\emptyset \vdash \mathtt{v}_0 ... \mathtt{v}_n : \mathtt{S}''_0\mathtt{S}''_1 ... \mathtt{S}''_n$ with $\mathtt{S}''_0\mathtt{S}''_1 ... \mathtt{S}''_n \leq \mathtt{S}'_0\mathtt{S}'_1 ... \mathtt{S}'_n$ and $\emptyset \vdash \mathtt{w}_1 : \mathtt{S}'''_1$, ..., $\emptyset \vdash \mathtt{w}_m : \mathtt{S}'''_m$ with $\mathtt{S}'''_l \leq \mathtt{S} \leq \mathtt{S}_0$ $(1 \leq l \leq m)$. We have two subcases.

- If $\mathtt{f}$ is a user-defined function, then from (C.2) we get (by reasoning as in the proof of case $\mathtt{d}(\mathtt{e}_1, ..., \mathtt{e}_n)$) $\emptyset \vdash \mathtt{u}_1 : \mathtt{S}''''_1$, ..., $\emptyset \vdash \mathtt{u}_m : \mathtt{S}''''_m$ for some $\mathtt{S}''''_l$ such that $\mathtt{S}''''_l \leq \mathtt{S}'$ $(1 \leq l \leq m)$.
- If $\mathtt{f}$ is a built-in function, then from (C.2) we get (by the semantics of built-in functions) $\emptyset \vdash \mathtt{u}_1 : \mathtt{S}''''_1$, ..., $\emptyset \vdash \mathtt{u}_m : \mathtt{S}''''_m$ for some $\mathtt{S}''''_l$ such that $\mathtt{S}''''_l \leq \mathtt{S}'$ $(1 \leq l \leq m)$.

In both cases $\mathtt{v} = \bigwedge\{\mathtt{v}_0, \mathtt{u}_1, ..., \mathtt{u}_m\}$ has a sort $\mathtt{S}''$ with $\mathtt{S}''' \leq \mathtt{S}$, i.e., $\emptyset \vdash \mathtt{v} : \mathtt{S}'''$ with $\mathtt{S}''' \leq \mathtt{S}$ holds. □

Restatement of Theorem 8.6 (Device computation sort preservation). *If $\overline{x} : \overline{S} \vdash e : S$, σ is a sensor mapping, $\overline{\theta} \in WSVT(\overline{x} : \overline{S}, e, S)$, $length(\overline{v}) = length(\overline{x})$, $\emptyset \vdash \overline{v} : \overline{S}'$, $\overline{S}' \leq \overline{S}$, and $\sigma; \overline{\theta} \vdash e[\overline{x} := \overline{v}] \Downarrow \theta$, then $\emptyset \vdash \rho(\theta) : S'$ for some S' such that $S' \leq S$.*

Proof. Straightforward by Lemma C.2, since $\theta \in WSVT(\overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{e}, \mathtt{S})$. □

Appendix D. Proof of Theorem 9.5

Lemma D.1 (Annotated sort of an expression)**.** *If $x_1 : S_1$ [?], $\overline{x} : \overline{S} \vdash e : S[\pi]$, then $x : S_1, \overline{x} : \overline{S} \vdash e : S$ and $\top_{key(S_1)} = \top_{key(S)}$.*

Proof. Straightforward by induction on the application of the annotated sort checking rules for expressions in Fig. 23. □

A pure expression $\mathtt{e}$ with free variables $\overline{\mathtt{x}}$ of sorts $\overline{\mathtt{S}}$ represents the pure function that for every $\overline{\mathtt{v}} \in [\![\overline{\mathtt{S}}]\!]$ returns the value $[\![\mathtt{e}[\overline{\mathtt{x}} := \overline{\mathtt{v}}]]\!]$. In the following we will write $fun(\mathtt{e})$ to denote such a function.

Lemma D.2 (Annotation soundness for expressions)**.** *If , $x_1 : S_1$ [?], $\overline{x} : \overline{S} \vdash e : S[\pi]$ and $\overline{v} \in [\![\overline{S}]\!]$, then*

(1) *if $\pi =$! then*
  - *$v <^1_{S_1} v'$ and $[\![fun(e)]\!](v, \overline{v}) = v'' \neq^1_S \top_S$ imply implies $v'' <^1_S [\![fun(e)]\!](v', \overline{v})$;*
  - *for all $v \in [\![S_1]\!] - \{\top_{S_1}\}$, $key(v) <_{key(S_1)} key([\![fun(e)]\!](v, \overline{v}))$;*

(2) *if $\pi \in \{$!, ?$\}$ then*
  - *$v \leq^1_{S_1} v'$ implies $[\![fun(e)]\!](v, \overline{v}) \leq^1_S [\![fun(e)]\!](v', \overline{v})$;*
  - *for all $v \in [\![S_1]\!]$, $key(v) \leq_{key(S_1)} key([\![fun(e)]\!](v, \overline{v}))$.*

Proof. By Lemma D.1 $[\![\mathtt{e}]\!]$ has sort $\mathtt{S}(\mathtt{S}_1\overline{\mathtt{S}})$ and either $key(\mathtt{S}_1) \leq^{\text{progressive}} key(\mathtt{S})$ or $key(\mathtt{S}) \leq^{\text{progressive}} key(\mathtt{S}_1)$. Recall that the annotated sort checking rules (in Fig. 23) and the evaluation rules (in Fig. 6 and Fig. 14) are syntax directed. By induction on the syntax of pure expressions $\mathtt{e}$. The base of the induction on $\mathtt{e}$ consist of two cases.

Case $\mathtt{x}$: Immediate by rule [A-VAR].

Case $\mathtt{v}$: Straightforward by rule [A-GVAL] and Proposition 7.5.

For the inductive step on $\mathtt{e}$, we show only the case for function application (all the other cases are straightforward by induction).

Case $\mathtt{f}(\mathtt{e}_1, \overline{\mathtt{e}})$: Then the premises of rule [A-FUN]:
- $\mathscr{A} \vdash \mathtt{e}_1 : \mathtt{S}'_1 [\pi'']$
- $|\mathscr{A}| \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}}'$
- $\mathtt{S}(\mathtt{S}''_1\overline{\mathtt{S}}'') [\pi'] \in ms(a\text{-}s\text{-}sigs(\mathtt{f}), \mathtt{S}'_1\overline{\mathtt{S}'})$

hold and $\pi = \pi'(\pi'')$. By the last premise, we have that π'-*prestabilising*$(\mathtt{f}, \mathtt{S}(\mathtt{S}''_1\overline{\mathtt{S}}''))$ holds. Then, the result follows straightforward by induction (using Definition 7.1). □

Restatement of Theorem 9.5 (Annotation soundness). *If* $\vdash \mathtt{D} : \overline{\mathtt{S}(\overline{\mathtt{S}})\,\mathtt{[}\pi\mathtt{]}}$ *holds, then* π*-prestabilising*$(f, \mathtt{S}(\overline{\mathtt{S}}))$ *holds for all* $\mathtt{S}(\overline{\mathtt{S}})\,\mathtt{[}\pi\mathtt{]} \in \overline{\mathtt{S}(\overline{\mathtt{S}})\,\mathtt{[}\pi\mathtt{]}}$.

Proof. Straighforward from rule [A-DEF] in Fig. 19 using Lemma D.2 and Proposition 7.4. □

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/2.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany