Predictable Self-Organization with Computational Fields Pt 2

Computational Fields: calculus, examples, self-stabilisation

Mirko Viroli [1] Jacob Beal [2]

[1] Alma Mater Studiorum—Università di Bologna, Italy
mirko.viroli@unibo.it

[2] BBN Technologies, USA
jakebeal@bbn.com

SASO 2014, London, 8/9/2014

Sub-Outline

1. Why a calculus of fields
2. The Field Calculus – "global viewpoint"
3. Semantic details – "local viewpoint"
4. Bootstrapping property verification: Self-stabilisation

Outline

1. Why a calculus of fields
2. The Field Calculus – "global viewpoint"
3. Semantic details – "local viewpoint"
4. Bootstrapping property verification: Self-stabilisation

Scope of the research

Towards a foundational approach

- Many languages/modes incorporate features of aggregate programming
  (see a survey in [Beal et al., 2013])
- No general formalisation approaches exist

Scope of the research

Towards a foundational approach

- Many languages/modes incorporate features of aggregate programming
  (see a survey in [Beal et al., 2013])
- No general formalisation approaches exist

Our research steps

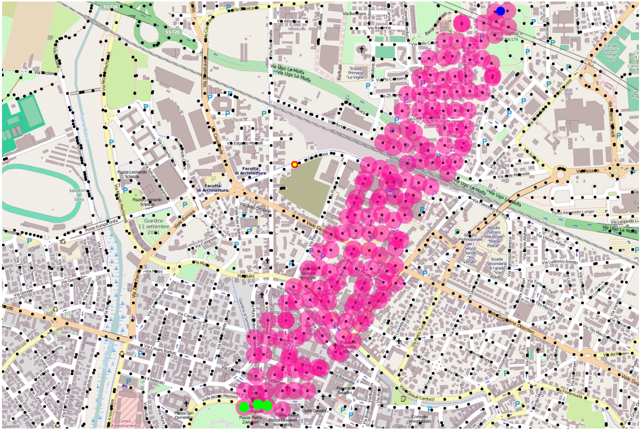
1. Identify a minimal set of ingredients of spatial computing
2. Devise a core calculus of computational fields [Viroli et al., 2013]
3. Provide a full formalisation in the proglangs/concurrency style
4. Isolate fragments enjoying certain properties

Expected impact

Perspective

- Paving the way towards advanced behavioural analysis
  - Sufficient conditions for self-stabilisation and density independence
  - Type-checking correctness properties
  - Formal local-to-global and global-to-local connections
- Devising tools for proper system engineering
  - building blocks, languages, patterns, APIs, infrastructure

An example: a “channel” deployed in physical space



- Computational field (or field): a mapping from nodes to values
- Can be: static, dynamic, or require time to stabilise
- Sometimes abstracted to a continuous space-time domain

How to achieve this structure in a self-organised way?

Local viewpoint

- What is the single-node behaviour?
- How should a node interact with neighbours and locally compute?

Global viewpoint

- What is the global structure that emerges, what are its properties?
- Can we create it compositionally?

How to achieve this structure in a self-organised way?

Local viewpoint

- What is the single-node behaviour?
- How should a node interact with neighbours and locally compute?

Global viewpoint

- What is the global structure that emerges, what are its properties?
- Can we create it compositionally?

Computing with fields as first-class abstractions

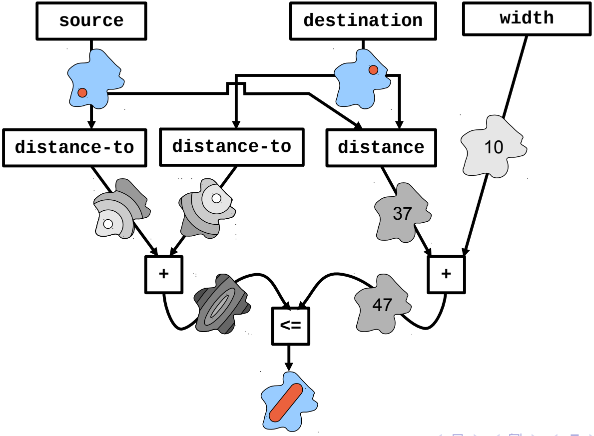
- "Inputs" (dynamically coming from sensors) are fields!
- "Outputs" (data produced, signal to actuators) are fields!
- Computation is a function from inputs to outputs as usual

$\Rightarrow$ .. use a functional style to manipulate field expressions

How would we create a "channel" compositionally?

Using a functional style and mathematical operators

$$channel(src, dest, width) =$$
$$dilate(distance\text{-}to(src) + distance\text{-}to(dest) \leq distance(src, dest), width)$$



From global back to local viewpoint

Working at the global level is key

- Provides a more abstract framework
- Captures desired concepts more directly
- Facilitates design, specification, understanding

We need to map back to locality

- How do we "compile" the global specification back in the single node behaviour?

A core calculus of computational fields

What do we need to tackle the problem solidly?

- A handy formalisation framework
- Built on top of known and well-studied techniques
- Quickly leading to tool implementation

A core calculus of computational fields

What do we need to tackle the problem solidly?

- A handy formalisation framework
- Built on top of known and well-studied techniques
- Quickly leading to tool implementation

The notion of core calculus

- a tiny language: syntax + operational semantics (+ typing)
- strives for the best tradeoff between:
  - compactness, simplicity (fewest constructs/concepts)
  - expressiveness (many program/behaviours)
- popular examples:
  - λ-calculus [Barendregt, 1984]: a core for functional programming
  - π-calculus [Milner, 1999]: a core for interactive programming
  - FJ [Igarashi et al., 1999]: a core for object-oriented programming

  $\Rightarrow$ virtually any programming constructs is formulated in that way..

The case of λ-calculus

Formalisation

- Syntax: lambdas, variables, application
- Semantics: one-step execution of function application

$$L ::= \lambda x.L \mid x \mid LL$$

$$(\lambda x.L)M \rightarrow L[M/x]$$

Outline

1. Why a calculus of fields
2. The Field Calculus – “global viewpoint”
3. Semantic details – “local viewpoint”
4. Bootstrapping property verification: Self-stabilisation

The calculus of computational fields [Viroli et al., 2013]

Main constructs

- Retains the functional style (recursive function definitions, one main)
- Built-in functions model sensor acquisition and any local algorithm
- Key computational field constructs
  - `rep` — Repetition: state evolving over time
  - `nbr` — Neighbouring: get information from neighbours
  - `if` — Restriction: a space-time branch

Other facts

- Syntax is LISP-like, in fact, a core of Proto [MIT Proto, 2012]
- Will use MIT Proto implementation for demoing
- Nodes all execute the same program, in asynchronous rounds
- Env. information (topology, sensors) extracted by built-in functions

Syntax

$\mathtt{e}$	$::=$		(field) expression:
		$\mathtt{l}$	local value (boolean, float, tuple ...)
	$\mid$	$\mathtt{x}$	variable
	$\mid$	$(\mathtt{b}\ \mathtt{e}_1\ \mathtt{e}_2\ \ldots\ \mathtt{e}_n)$	functional composition ($\mathtt{b}$ is built-in)
	$\mid$	$(\mathtt{f}\ \mathtt{e}_1\ \mathtt{e}_2\ \ldots\ \mathtt{e}_n)$	function call ($\mathtt{f}$ is user-defined)
	$\mid$	$(\textbf{rep}\ \mathtt{x}\ \mathtt{l}\ \mathtt{e})$	time evolution
	$\mid$	$(\textbf{nbr}\ \mathtt{e})$	neighbourhood field construction
	$\mid$	$(\textbf{if}\ \mathtt{e}_b\ \mathtt{e}_t\ \mathtt{e}_f)$	restriction
$\mathtt{F}$	$::=$	$(\textbf{def}\ \mathtt{f}(\mathtt{x}_1\ \mathtt{x}_2\ \ldots\ \mathtt{x}_n)\ \mathtt{e})$	user-defined function

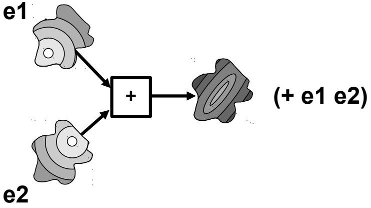
Functional composition: (b e_1 ... e_n)

Goal: extending standard computation mechanisms to whole fields

- b is a built-in operator applied to e_1 ... e_n pointwise (node per node)
- a means to perform local operations: algorithms/sensing/acting
- they are pervasively used, shall use the following color

Basic example

- `(+ e1 e2)`: sums the two fields e1 and e2



Built-in functions as environment bindings

Examples

- `(red 1)` send value 1 to `red` actuator (a led)
- `(sense "temperature")` gives a field of temperature values
- `(sense 1)` gives a field of values from sensor n. 1
- `(dt)` gives the length of last round in each node
- `(mid)` gives the field of device identifiers

Built-in functions and tuples

How do we deal with more articulated data types

- they are still seen as locals
- built-in functions are ADT operations, they can be used to create and operate on such complex values

Examples using tuples

- `(tup 10 20)` gives everywhere a tuple of two values, 10 and 20
- `(1st (tup 10 20))` accesses first component, i.e., 10
- `(2nd (tup 10 20))` accesses second component, i.e., 20

Built-in functions and "special" neighbourhood fields

A peculiar data-type we handle is a neighbourhood field ϕ

A map $\{\sigma_1 \mapsto \mathtt{l}_1, \ldots, \sigma_n \mapsto \mathtt{l}_n\}$ from neighbourhood to some local value

- its domain never escapes a node's neighbourhood
- it can't be the final result of computation, just an intermediate one
- some special built-in sensor could return one such field (`nbr-*`)
- some built-in function (`*-hood`,`*-hood+`) can turn it into a value

Examples using fields

- `(nbr-range)` gives a map ϕ from neighbours to their distance
- `(min-hood+ (nbr-range))` gives the minimum of neighbour distances (+ means "myself excluded")
- `(< (min-hood+ (nbr-range)) 5)` gives nodes with some short proximity

Function calls: (f e_1 ... e_n)

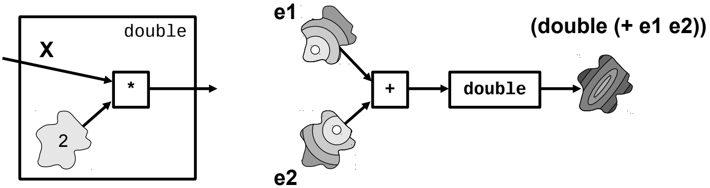
Goal: providing standard name abstraction and recursive behaviour

- f is the name of a function to be defined as: (**def** f($x_1 \ldots x_n$) e_{body})
- call-by-value semantics, possibly with recursion
- for user-defined functions shall use the following color

Example

Def `(def double(x) (* x 2))`

Use `(double (+ e1 e2))`: doubles the field `(+ e1 e2)`



Syntax

e	::=		(field) expression:
		l	local value (boolean, float, tuple ...)
	|	x	variable
	|	(b e_1 e_2 ... e_n)	functional composition (b is built-in)
	|	(f e_1 e_2 ... e_n)	function call (f is user-defined)
	|	(**rep** x l e)	time evolution
	|	(**nbr** e)	neighbourhood field construction
	|	(**if** e_b e_t e_f)	restriction
F	::=	(**def** f(x_1 x_2 ... x_n) e)	user-defined function

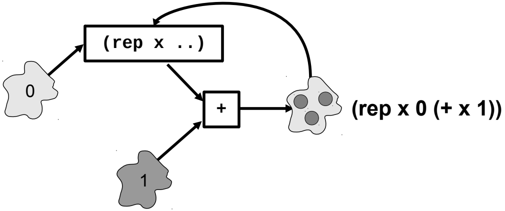
Field construct n.1: Time evolution (**rep** `x` `l` `e`)

Goal: supporting field evolution over time

- Initially it is globally `l` (or any expression..)
- At each new step a point in space updates to (local value of) `e`
- `e` can mention `x`, which stands for "previous field (state)"

Example

- `(rep x 0 (+ x 1))`: counts the number of rounds in each device
- `(rep t 0 (+ t (dt)))`: the field of time



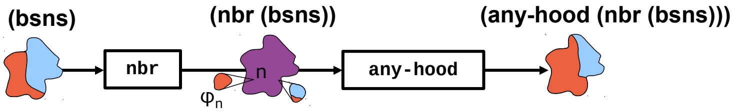
Field construct n.2: Neigh. field construction (**nbr** e)

Goal: enabling declarative node-to-node interaction

- The resulting field maps each node n to a neighbourhood field ϕ_n
- ϕ_n maps neighbours of n to their local value of e
- Is to be flattened by a `*-hood` built-in operator

Example

- Assume bsns be a special operator modelling a boolean sensor
- `(any-hood (`**nbr**` (bsns)))`: has any neighbour positive `bsns`?



Few field constructions examples

```
;; the field of number of neighbours
(def count-neighbours () (sum-hood (nbr 1)))

;; average distance of neighbours
(/ (sum-hood (nbr-range)) (count-neighbours))

;; most connected neighbour's id?
(def most-connected-neighbour ()
    (2nd (max-hood (nbr (tup (count-neighbours) (mid)))))
)

;; what's the highest connection in the network?
???
```

Neighbourhood chaining: nesting nbr into rep

```
;; what's the greatest value of F?
(def goss-max (F) (rep x F (max-hood (nbr x))))

;; which node is the highest connected?
(def most-connected-node ()
    (2nd (goss-max (tup (count-neighbours) (mid)))))

;; what's the minimum hop-count distance to (any) source?
(def hop-count (source)
    (rep x (inf) (mux source 0 (min-hood+ (+ 1 (nbr x))))))

;; distance-to, aka, gradient --> note it is robust and flexible!!
(def distance-to (source)
  (rep d (inf) (mux source 0 (min-hood+ (+ (nbr d) (nbr-range))))))
```

Building a channel

```
;; broadcast, aka gradcast, broadcasting the value of f at src
(def broadcast (src f)
    (rep t (tup (inf) f)
           (mux src (tup 0 f)
                    (min-hood+ (tup (+ (nbr-range) (1st (nbr t)))
                                    (2nd (nbr t))))))))

;; minimum distance between a and b
(def distance (a b)
    (2nd (broadcast b (distance-to a))))

;; channel between src and dest of thickness width
(def channel (src dest width)
    (< (+ (distance-to src) (distance-to dest))
       (+ width (distance src dest))))
```

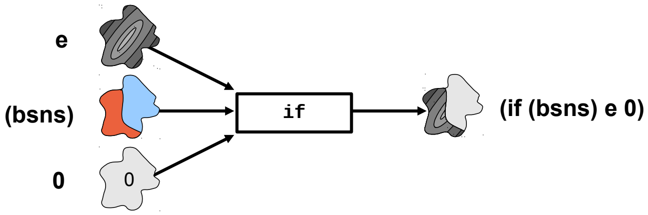
Field construct n.3: Restriction (**if** e_b e_t e_f)

Goal: providing a distributed branch

- e_b should be a boolean field (defining two restricted domains)
- Construct e_t where e_b is positive, and e_f where e_b is negative
- Not a mere superposition of e_t and e_f, but a true domain restriction!
  - i.e., `nbrs` inside e_t should not escape into where e_b is false

Example

- `(if (bsns) e 0)` creates `e` where `bsns` is true, and 0 elsewhere

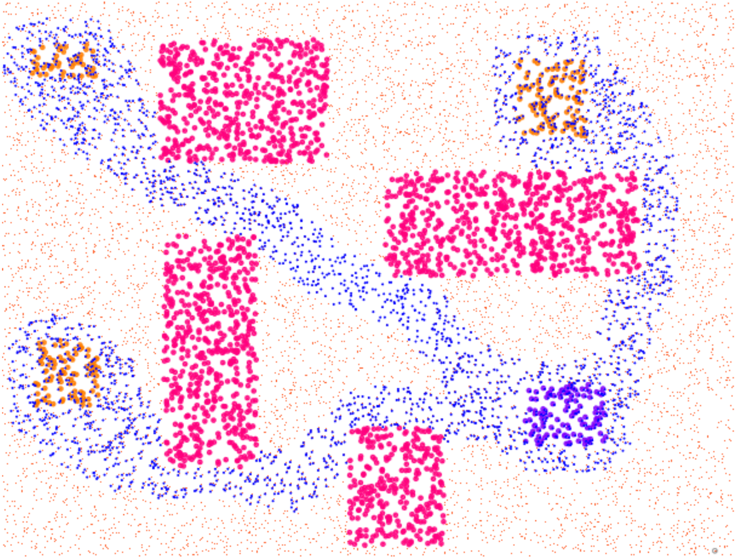


Channel avoiding obstacles

```
;; automatically circumventing the obstacle
(def channel-avoiding-obstacle (obstacle source destination width)
    (if (not obstacle) (channel source destination width) 0))

;; mux applies the obstacle
;; but it does not interfere with channel computation
(def wrong-channel-avoiding-obstacles (obstacle source destination width)
    (mux (not obstacle) (channel source destination width) 0))
```

Channel in action – a simulation in Proto



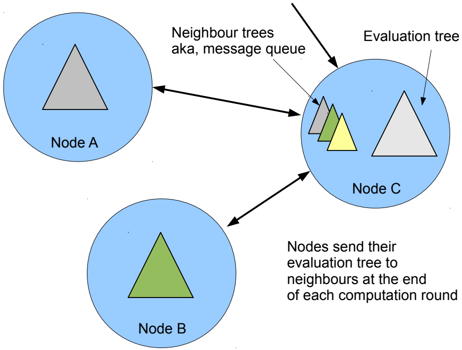
Outline

1. Why a calculus of fields
2. The Field Calculus – "global viewpoint"
3. Semantic details – "local viewpoint"
4. Bootstrapping property verification: Self-stabilisation

Key aspects of the semantics: network model

Network model

- A node has a state S (evaluation tree) updated at asynchronous rounds
- At the end of the round, S is spread to the (current) neighbourhood
- State is updated “against” the neighbour states just received



Key aspects of the semantics: node (abstract) model

Inside a node: the evaluation tree S

- S is an annotated version of the expression to evaluate
  - annotations are used to keep track of the next evaluation site
  - some annotation are persistent, and used to interact in space/time
- S may dynamically expand due to (recursive) calls
- field constructs semantics impact shape of annotations
  - `rep`: last result is stored in an annotation and reminded at next round
  - `nbr`: observes annotations in the same position in neighbour trees
  - `if`: discards the neighbour trees which took a different branch

Main issue (and contribution wrt previous formalisation attempts)

Accommodate the interplay between `rep`, `if` and `nbr` – even in the presence of (recursive) calls

A node state is an annotated evaluation tree

Round 1 $\texttt{(rep x 0 (+ }\underline{\texttt{x}}\texttt{ 1))} \rightarrow \texttt{(rep x 0 (+ x}\cdot\texttt{0 }\underline{\texttt{1}}\texttt{))} \rightarrow$
$\texttt{(rep x 0 }\underline{\texttt{(+ x}\cdot\texttt{0 1}\cdot\texttt{1)}}\texttt{)} \rightarrow \underline{\texttt{(rep x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}}$
$\rightarrow \texttt{(rep}^{1}\texttt{ x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}\cdot\texttt{1}$

A node state is an annotated evaluation tree

Round 1 $\;$ $\texttt{(rep x 0 (+ }\underline{\texttt{x}}\texttt{ 1))} \rightarrow \texttt{(rep x 0 (+ x}\cdot\textcolor{red}{\texttt{0}}\texttt{ }\underline{\texttt{1}}\texttt{))} \rightarrow$
$\texttt{(rep x 0 }\underline{\texttt{(+ x}\cdot\texttt{0 1}\cdot\textcolor{red}{\texttt{1}}\texttt{)}}\texttt{)} \rightarrow \underline{\texttt{(rep x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\textcolor{red}{\texttt{1}}\texttt{)}}$
$\rightarrow \texttt{(rep}^{\textcolor{blue}{1}}\texttt{ x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}\cdot\textcolor{red}{\texttt{1}}$

Round 2 $\;$ $\texttt{(rep}^{\textcolor{blue}{1}}\texttt{ x 0 (+ x 1))} \rightarrow .. \rightarrow \texttt{(rep}^{\textcolor{blue}{2}}\texttt{ x 0 (+ x}\cdot\texttt{1 1}\cdot\texttt{1)}\cdot\texttt{2)}\cdot\textcolor{red}{\texttt{2}}$

A node state is an annotated evaluation tree

Round 1 $\texttt{(rep x 0 (+ } \underline{\texttt{x}} \texttt{ 1))} \rightarrow \texttt{(rep x 0 (+ x}\cdot\texttt{0 } \underline{\texttt{1}}\texttt{))} \rightarrow$
$\texttt{(rep x 0 } \underline{\texttt{(+ x}\cdot\texttt{0 1}\cdot\texttt{1)}}\texttt{)} \rightarrow \underline{\texttt{(rep x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}}$
$\rightarrow \texttt{(rep}^{1}\texttt{ x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}\cdot\texttt{1}$

Round 2 $\texttt{(rep}^{1}\texttt{ x 0 (+ x 1))} \rightarrow .. \rightarrow \texttt{(rep}^{2}\texttt{ x 0 (+ x}\cdot\texttt{1 1}\cdot\texttt{1)}\cdot\texttt{2)}\cdot\texttt{2}$

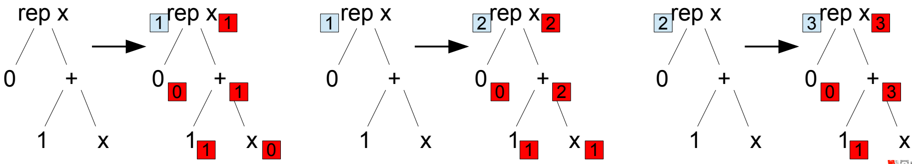
Round 3 $\texttt{(rep}^{2}\texttt{ x 0 (+ x 1))} \rightarrow .. \rightarrow \texttt{(rep}^{3}\texttt{ x 0 (+ x}\cdot\texttt{2 1}\cdot\texttt{1)}\cdot\texttt{3)}\cdot\texttt{3}$

A node state is an annotated evaluation tree

Round 1 $\texttt{(rep x 0 (+ }\underline{\texttt{x}}\texttt{ 1))} \rightarrow \texttt{(rep x 0 (+ x}\cdot\texttt{0 }\underline{\texttt{1}}\texttt{))} \rightarrow$
$\texttt{(rep x 0 }\underline{\texttt{(+ x}\cdot\texttt{0 1}\cdot\texttt{1)}}\texttt{)} \rightarrow \underline{\texttt{(rep x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}}$
$\rightarrow \texttt{(rep}^{1}\texttt{ x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}\cdot\texttt{1}$

Round 2 $\texttt{(rep}^{1}\texttt{ x 0 (+ x 1))} \rightarrow .. \rightarrow \texttt{(rep}^{2}\texttt{ x 0 (+ x}\cdot\texttt{1 1}\cdot\texttt{1)}\cdot\texttt{2)}\cdot\texttt{2}$

Round 3 $\texttt{(rep}^{2}\texttt{ x 0 (+ x 1))} \rightarrow .. \rightarrow \texttt{(rep}^{3}\texttt{ x 0 (+ x}\cdot\texttt{2 1}\cdot\texttt{1)}\cdot\texttt{3)}\cdot\texttt{3}$



Evaluation aligns with neighbours to support nbr

Neighbour trees:

$$\begin{array}{lcl} \sigma_1 & \mapsto & \texttt{(min-hood (nbr (sns)}\cdot 4\texttt{)}\cdot\phi_1\texttt{)}\cdot 4 \\ \sigma_2 & \mapsto & \texttt{(min-hood (nbr (sns)}\cdot 9\texttt{)}\cdot\phi_2\texttt{)}\cdot 9 \end{array}$$

Evaluation (assume here sns gives 7):

$$\begin{array}{l} \texttt{(min-hood (nbr }\underline{\texttt{(sns)}}\texttt{))} \to \\ \texttt{(min-hood }\underline{\texttt{(nbr (sns)}\cdot 7\texttt{)}}\texttt{)} \to \\ \underline{\texttt{(min-hood (nbr (sns)}\cdot 7\texttt{)}\cdot(\sigma \mapsto 7, \sigma_1 \mapsto 4, \sigma_2 \mapsto 9)\texttt{)}} \to \\ \texttt{(min-hood (nbr (sns)}\cdot 7\texttt{)}\cdot(\sigma \mapsto 7, \sigma_1 \mapsto 4, \sigma_2 \mapsto 9)\texttt{)}\cdot 4 \end{array}$$

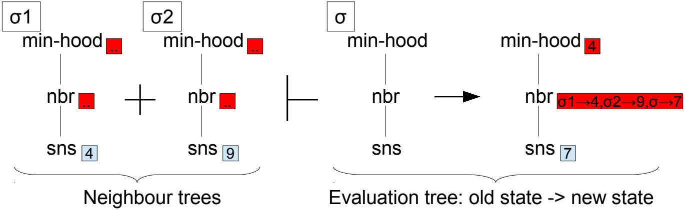
Evaluation aligns with neighbours to support nbr

Neighbour trees:

$$\sigma_1 \mapsto \texttt{(min-hood (nbr (sns)}\cdot 4\texttt{)}\cdot\phi_1\texttt{)}\cdot 4$$
$$\sigma_2 \mapsto \texttt{(min-hood (nbr (sns)}\cdot 9\texttt{)}\cdot\phi_2\texttt{)}\cdot 9$$

Evaluation (assume here sns gives 7):

$$\texttt{(min-hood (nbr }\underline{\texttt{(sns)}}\texttt{))} \rightarrow$$
$$\texttt{(min-hood }\underline{\texttt{(nbr (sns)}\cdot 7\texttt{)}}\texttt{)} \rightarrow$$
$$\underline{\texttt{(min-hood (nbr (sns)}\cdot 7\texttt{)}\cdot(\sigma \mapsto 7, \sigma_1 \mapsto 4, \sigma_2 \mapsto 9))} \rightarrow$$
$$\texttt{(min-hood (nbr (sns)}\cdot 7\texttt{)}\cdot(\sigma \mapsto 7, \sigma_1 \mapsto 4, \sigma_2 \mapsto 9))\cdot 4$$



Restriction prevents escaping a domain

Neighbour trees:

$$\sigma_1 \mapsto \texttt{(if (bsns)}{\cdot}f\ \texttt{(min-hood (nbr (sns))) 0}{\cdot}0\texttt{)}{\cdot}0$$

$$\sigma_2 \mapsto \texttt{(if (bsns)}{\cdot}t\ \texttt{(min-hood (nbr (sns)}{\cdot}9\texttt{)}{\cdot}\phi_2\ 0\texttt{))}{\cdot}9$$

Evaluation (assume here sns gives 7 and bsns gives t):

$$\texttt{(if } \underline{\texttt{(bsns)}} \texttt{ (min-hood (nbr (sns))) 0)} \rightarrow$$

$$\texttt{(if (bsns)}{\cdot}t\ \texttt{(min-hood (nbr } \underline{\texttt{(sns)}}\texttt{)) 0)} \rightarrow \ldots \rightarrow$$

$$\underline{\texttt{(if (bsns)}{\cdot}t\ \texttt{(min-hood (nbr (sns)}{\cdot}7\texttt{)}{\cdot}(\sigma \mapsto 7, \sigma_2 \mapsto 9)\texttt{))}{\cdot}7\ \texttt{0)}}$$

Neighbour trees:

$$\sigma_1 \mapsto \texttt{(if (bsns)}{\cdot}\color{blue}{f}\ \texttt{(min-hood (nbr (sns))) 0}{\cdot}0\texttt{)}{\cdot}0$$

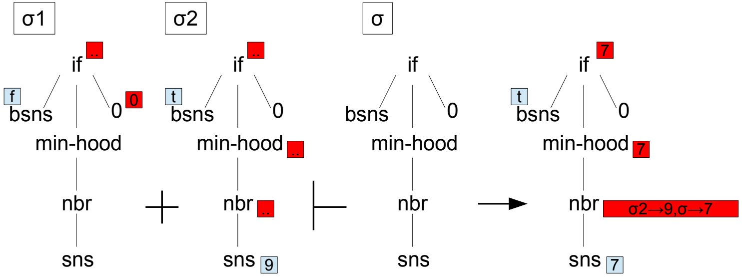
$$\sigma_2 \mapsto \texttt{(if (bsns)}{\cdot}\color{blue}{t}\ \texttt{(min-hood (nbr (sns)}{\cdot}\color{blue}{9}\texttt{)}{\cdot}\phi_2\ \texttt{0))}{\cdot}9$$

Evaluation (assume here sns gives 7 and bsns gives t):

$$\texttt{(if } \underline{\texttt{(bsns)}}\texttt{ (min-hood (nbr (sns))) 0)} \rightarrow$$

$$\texttt{(if (bsns)}{\cdot}\color{blue}{t}\ \texttt{(min-hood (nbr } \underline{\texttt{(sns)}}\texttt{)) 0)} \rightarrow \ldots \rightarrow$$

$$\underline{\texttt{(if (bsns)}{\cdot}\color{blue}{t}\ \texttt{(min-hood (nbr (sns)}{\cdot}\color{blue}{7}\texttt{)}{\cdot}(\sigma \mapsto 7, \sigma_2 \mapsto 9)\texttt{))}{\cdot}\color{red}{7}\ \texttt{0)}}$$



Recursively, neighbour trees that are not aligned are temporaneously discarded

Function calls dynamically expand the evaluation tree

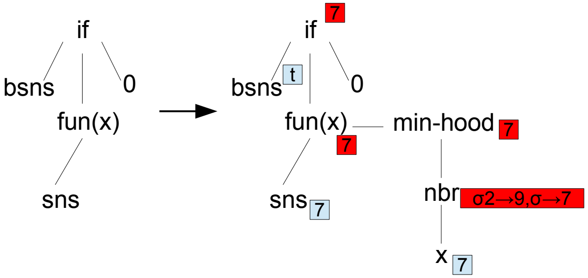
Take definition (**def** `fun (x) (min-hood (nbr x))`)..
When evaluating `(if (bsns) (fun sns) 0)`, function body is expanded:

$$\texttt{(if }\underline{\texttt{(bsns)}}\texttt{ (fun sns) 0)} \rightarrow \texttt{(if (bsns)}\cdot t\texttt{ (fun }\underline{\texttt{(sns)}}\texttt{) 0)} \rightarrow$$
$$\texttt{(if (bsns)}\cdot t\ \underline{\texttt{(fun (sns)}\cdot 7\texttt{)}}\texttt{ 0)} \rightarrow$$
$$\texttt{(if (bsns)}\cdot t\texttt{ (fun}^{\texttt{(min-hood (nbr }\underline{\texttt{x}}\texttt{))}}\texttt{ (sns)}\cdot 7\texttt{) 0)} \rightarrow$$
$$\texttt{(if (bsns)}\cdot t\texttt{ (fun}^{\texttt{(min-hood }\underline{\texttt{(nbr x}\cdot 7\texttt{)}}\texttt{)}}\texttt{ (sns)}\cdot 7\texttt{) 0)} \rightarrow \ldots$$

Function calls dynamically expand the evaluation tree

Take definition (**def** `fun (x) (min-hood (nbr x))`)..
When evaluating `(if (bsns) (fun sns) 0)`, function body is expanded:

$$\texttt{(if }\underline{\texttt{(bsns)}}\texttt{ (fun sns) 0)} \rightarrow \texttt{(if (bsns)}\cdot t\texttt{ (fun }\underline{\texttt{(sns)}}\texttt{) 0)} \rightarrow$$
$$\texttt{(if (bsns)}\cdot t\ \underline{\texttt{(fun (sns)}\cdot 7\texttt{)}}\texttt{ 0)} \rightarrow$$
$$\texttt{(if (bsns)}\cdot t\texttt{ (fun}^{\texttt{(min-hood (nbr }\underline{\texttt{x}}\texttt{))}}\texttt{ (sns)}\cdot 7\texttt{) 0)} \rightarrow$$
$$\texttt{(if (bsns)}\cdot t\texttt{ (fun}^{\texttt{(min-hood }\underline{\texttt{(nbr x}\cdot 7\texttt{)}}\texttt{)}}\texttt{ (sns)}\cdot 7\texttt{) 0)} \rightarrow \ldots$$



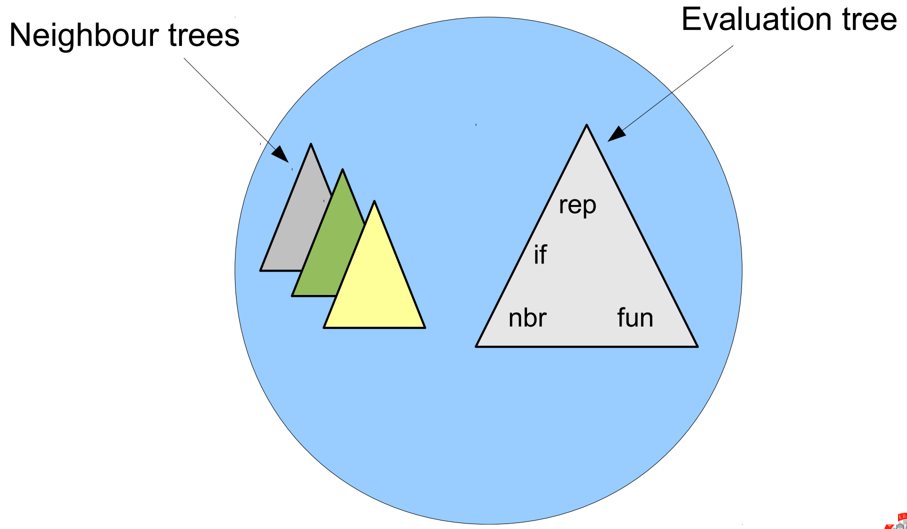
Expansion at function call, contraction on non-taken branches

The pillars of the operational semantics

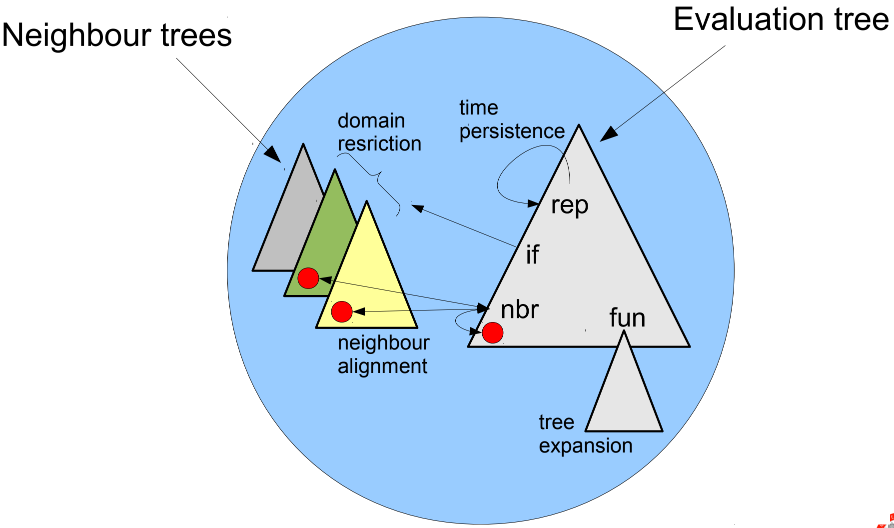
Elements

- A node's computation is about annotating the evaluation tree
- Such a tree is spread to neighbours after being cleaned up
- Neighbour trees affect evaluation
- Annotations to **nbr**
  - persist over "space" (sent to neighbours)
  - used to remotely reconstruct a neighbourhood field at **nbr** sites
- Annotations to **if**
  - persist over "space" (sent to neighbours)
  - used to filter out neighbours that do not match at **if** sites
- Annotations to **rep**
  - persist over "time" (remembered at next round)
  - used to recompute at **rep** sites

Tree evaluation: pictorial semantics



Tree evaluation: pictorial semantics



The operational semantics of node model

Runtime Expression Syntax:

$e ::= a{\cdot}\mathring{v}$	runtime expression (rte)
$a ::= \mathtt{x} \mid v \mid (\mathtt{nbr}\ e) \mid (\mathtt{if}\ e\ e\ e) \mid (\mathtt{rep}^{\mathtt{s}}\ \mathtt{x}\ \mathtt{w}\ e) \mid (\mathtt{f}^{\mathtt{s}}\ \overline{e}) \mid (\mathtt{o}\ \overline{e})$	auxiliary rte
$v ::= \mathtt{l} \mid \phi$	runtime value
$\mathtt{s} ::= \mathring{a}$	superscript
$\mathtt{w} ::= \mathtt{x} \mid \mathtt{l}$	variable or local value
$\phi ::= \overline{\sigma} \mapsto \overline{\mathtt{l}}$	field value
$\Theta ::= \overline{\sigma} \mapsto \overline{e}$	tree environment
$\Gamma ::= \overline{\mathtt{x}} := \overline{v}$	variable environment

Congruence Contexts:

$$\mathbb{C} ::= (\mathtt{nbr}\ [\,]) \mid (\mathtt{f}^{\mathtt{s}}\ \overline{e}\ [\,]\ \overline{e}) \mid (\mathtt{o}\ \overline{e}\ [\,]\ \overline{e}) \mid (\mathtt{if}\ [\,]\ e\ e) \mid (\mathtt{if}\ a{\cdot}t\ [\,]\ e) \mid (\mathtt{if}\ a{\cdot}f\ e\ [\,])$$

Alignment contexts:

$$\mathbb{A} ::= \mathbb{C} \mid (\mathtt{rep}^{\mathtt{s}}\ \mathtt{x}\ \mathtt{w}\ [\,]) \mid (\mathtt{f}^{[\,]}\ \overline{a{\cdot}v})$$

Auxiliary functions:

$$\begin{aligned}
\pi_{\mathbb{A}}(\Theta,\Theta') &= \pi_{\mathbb{A}}(\Theta),\pi_{\mathbb{A}}(\Theta') \\
\pi_{\mathbb{A}}(\sigma \mapsto (\mathbb{A}'[e]){\cdot}v) &= \sigma \mapsto e \qquad \text{if } \mathbb{A}' :: \mathbb{A} \\
\pi_{\mathbb{A}}(\sigma \mapsto e) &= \bullet \qquad \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathtt{s} \rhd a &= a \\
\mathtt{s} \rhd \circ &= \mathtt{s}
\end{aligned}$$

$$\begin{aligned}
(\mathtt{nbr}\ [\,]) &:: (\mathtt{nbr}\ [\,]) \\
(\mathtt{f}^{\mathtt{s}'}\ e'_1 \ldots e'_{i-1}\ [\,]\ e'_{i+1} \ldots e'_n) &:: (\mathtt{f}^{\mathtt{s}}\ e_1 \ldots e_{i-1}\ [\,]\ e_{i+1} \ldots e_n) \\
(\mathtt{o}\ e'_1 \ldots e'_{i-1}\ [\,]\ e'_{i+1} \ldots e'_n) &:: (\mathtt{o}\ e_1 \ldots e_{i-1}\ [\,]\ e_{i+1} \ldots e_n) \\
(\mathtt{if}\ [\,]\ e'_1\ e'_2)) &:: (\mathtt{if}\ [\,]\ e_1\ e_2)) \\
(\mathtt{if}\ a'{\cdot}t\ [\,]\ e') &:: (\mathtt{if}\ a{\cdot}t\ [\,]\ e) \\
(\mathtt{if}\ a'{\cdot}f\ e'\ [\,]\ ) &:: (\mathtt{if}\ a{\cdot}f\ e\ [\,]\ ) \\
(\mathtt{rep}^{\mathtt{s}'}\ \mathtt{x}\ \mathtt{w}\ [\,]) &:: (\mathtt{rep}^{\mathtt{s}}\ \mathtt{x}\ \mathtt{w}\ [\,]) \\
(\mathtt{f}^{[\,]}\ e'_1 \ldots e'_n) &:: (\mathtt{f}^{[\,]}\ e_1 \ldots e_n)
\end{aligned}$$

Reduction Rules:

$$\text{[THEN]}\ \frac{}{\Theta;\Gamma \vdash (\mathtt{if}\ a{\cdot}t\ a{\cdot}\mathtt{l}\ e) \to (\mathtt{if}\ a{\cdot}t\ a{\cdot}\mathtt{l}\ |e|){\cdot}\mathtt{l}}$$

$$\text{[VAL]}\ \frac{}{\Theta;\Gamma \vdash v \to v{\cdot}v}$$

$$\text{[ELSE]}\ \frac{}{\Theta;\Gamma \vdash (\mathtt{if}\ a{\cdot}f\ e\ a{\cdot}\mathtt{l}) \to (\mathtt{if}\ a{\cdot}f\ |e|\ a{\cdot}\mathtt{l}){\cdot}\mathtt{l}}$$

$$\text{[VAR]}\ \frac{}{\Theta;\Gamma \vdash \mathtt{x} \to \mathtt{x}{\cdot}\Gamma(\mathtt{x})|_{dom(\Theta),\varepsilon(\mathtt{self})}}$$

$$\text{[CONG]}\ \frac{\pi_{\mathbb{C}}(\Theta);\Gamma \vdash a \to e}{\Theta;\Gamma \vdash \mathbb{C}[a] \to \mathbb{C}[e]}$$

$$\text{[NBR]}\ \frac{\pi_{(\mathtt{nbr}\ [\,])}(\Theta) = \overline{\sigma} \mapsto \overline{a}{\cdot}\overline{\mathtt{l}} \qquad \phi = (\overline{\sigma} \mapsto \overline{\mathtt{l}}, \varepsilon(\mathtt{self}) \mapsto \mathtt{l})}{\Theta;\Gamma \vdash (\mathtt{nbr}\ a{\cdot}\mathtt{l}) \to (\mathtt{nbr}\ a{\cdot}\mathtt{l}){\cdot}\phi}$$

$$\text{[REP]}\ \frac{\pi_{(\mathtt{rep}^{\mathring{\mathtt{l}}}\ \mathtt{x}\ \mathtt{w}\ [\,])}(\Theta);\Gamma,(\mathtt{x} := (\Gamma(\mathtt{w}) \rhd \mathring{\mathtt{l}})) \vdash a \to a'{\cdot}\mathring{v}}{\Theta;\Gamma \vdash (\mathtt{rep}^{\mathring{\mathtt{l}}}\ \mathtt{x}\ \mathtt{w}\ a) \to (\mathtt{rep}^{\mathring{\mathtt{l}} \rhd \mathring{v}}\ \mathtt{x}\ \mathtt{w}\ a'{\cdot}\mathring{v}){\cdot}\mathring{v}}$$

$$\text{[OP]}\ \frac{}{\Theta;\Gamma \vdash (\mathtt{o}\ \overline{a{\cdot}v}) \to (\mathtt{o}\ \overline{a{\cdot}v}){\cdot}\varepsilon(\mathtt{o},\overline{v})}$$

$$\text{[FUN]}\ \frac{\pi_{(\mathtt{f}^{[\,]}\ \overline{a{\cdot}v})}(\Theta);(args(\mathtt{f}) := \overline{v}) \vdash (body(\mathtt{f}) \rhd \mathtt{s}) \to a{\cdot}\mathring{v}}{\Theta;\Gamma \vdash (\mathtt{f}^{\mathtt{s}}\ \overline{a{\cdot}v}) \to (\mathtt{f}^{a}\ \overline{a{\cdot}v}){\cdot}\mathring{v}}$$

Outline

1. Why a calculus of fields
2. The Field Calculus – “global viewpoint”
3. Semantic details – “local viewpoint”
4. Bootstrapping property verification: Self-stabilisation

Bootstrapping property verification

What is the field calculus good at?

- Giving feedbacks on existing/new models/implementations
- Helping designing new features (e.g., typing, higher-order functions)
- Assisting development of surface languages, APIs

$\Rightarrow$ Predicting the behaviour of fragments of the language

Bootstrapping property verification

What is the field calculus good at?

- Giving feedbacks on existing/new models/implementations
- Helping designing new features (e.g., typing, higher-order functions)
- Assisting development of surface languages, APIs

$\Rightarrow$ Predicting the behaviour of fragments of the language

History and Roadmap

- PAST: Started in SAPERE EU FP7 project [Zambonelli et al., 2011]
  - self-stabilisation for a rule-based field language
- PRESENT: A self-stabilisation result for a fragment of the field calculus achieved [Viroli and Damiani, 2014]
- PRESENT: Universality of field calculus [Beal et al., 2014]
- ONGOING: Density-independence / approximability
- FUTURE: Extending the fragments of investigation

Self-stabilisation

Traditional self-stabilisation [Dolev, 2000]

A distributed system that is self-stabilising will end up in a *correct* state (in finite steps) no matter what state it is initialised with.

Superstabilisation

A distributed system is superstabilising if it is self-stabilising and it recovers *fast* from (single) topological changes.

Considerations for the field-calculus

- Self-stabilisation is of course a key fault-tolerant property
- The definition should somehow be adapted to work with fields
- One still needs to (implicitly) define what are the correct states
- Knowing that (and which) correct states are reached is indeed about predictability!
- What about: `(def flip (a b) (rep x a (- (+ a b) x)))`

Self-stabilisation for fields [Viroli and Damiani, 2014]

We considered a notion of (*unique*,*super-*) self-stabilisation

If a field expression is self-stabilising, then it necessarily reaches a unique configuration (unique fixpoint), recovering from *any* change

- assume execution of rounds is asynchronous but *fair*
- fixpoint is independent of initial state
- fixpoint is reached in finite time
- fixpoint depends on environment E (sensors/topology)
- if E (arbitrarily) changes the system recovers

Implication

Any self-stabilising field expression e has denotational semantics:

$$\phi_{final} = \phi_E(e)$$

..i.e., a program e has predictable global/final outcome ϕ_{final}

Sufficient conditions: a self-stabilising fragment

Ideas

- Generalise over a key block, the hop-count distance (gradient):

```
(rep x (inf) (mux source 0 (min-hood+ (+ 1 (nbr x)))))
```

- **nbr** should stay inside a ***-hood**, itself inside a **rep**
- ***-hood** should implement an order-independent function (e.g. min)
- Should generalise over the "+ 1" function
- Should guarantee that propagation terminates
- Strict layering of self-stabilising fields is self-stabilising

Some preliminaries

Rework a `distance-to` (gradient) specification

⇒ `(rep x (inf) (mux source 0 (min-hood+ (+ 1 (nbr x)))))`

- .. assume source is 0 on sources, and ∞ elsewhere

⇒ `(rep x (inf) (min source (min-hood+ (+ 1 (nbr x)))))`

- .. generalise over function $+$, which could also use sensor-dependent expressions

⇒
```
(def spr-aggr (E0 FUN E1 ..  En)
   (rep x (inf)
          (min E0 (FUN (min-hood+ x) E1 ..  En)))))
```

- .. abstract this block into a syntactic construct

⇒ $\{e_0 : g(@, e_1, \ldots, e_n)\}$

The Syntax – 3 main ingredients

e	::=	$\mathtt{x}$ |	variable
		$\mathtt{v}$ |	value
		s |	sensor
		$\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ |	pointwise functional composition
		$\{\mathtt{e} : \mathtt{g}(@, \mathtt{e}_1, \ldots, \mathtt{e}_n)\}$	spreading-aggregation (`nbr`,`rep`,`min-hood+`)
g	::=	$\mathtt{f}$ |	user-defined function
		$\mathtt{b}$	built-in function

Additional constraints

- User-defined function definitions have no cycles
- Built-in functions are environment independent
- Basic typing properties are satisfied

Stabilising progressions

A function $\mathtt{g}$ over type $\mathtt{T}$ is a *stabilising progression* if:

(i) $\mathtt{g}$ is a *pure operator* — it calls no sensor or spreading-aggregation;

(ii) $\mathtt{T}$ is *locally noetherian* — $[\![\mathtt{T}]\!]$ is equipped with a total order relation $\leq_{\mathtt{T}}$, and for every element $\mathtt{v} \in [\![\mathtt{T}]\!]$, there are no infinite ascending chains of elements $\mathtt{v}_0 <_{\mathtt{T}} \mathtt{v}_1 <_{\mathtt{T}} \mathtt{v}_2 \cdots$ such that (for every $n \geq 0$) $\mathtt{v}_n <_{\mathtt{T}} \mathtt{v}$;

(iii) $\mathtt{g}$ is *monotone* in its first argument — $\mathtt{v} \leq_{\mathtt{T}} \mathtt{v}'$ implies $[\![\mathtt{g}]\!](\mathtt{v}, \overline{\mathtt{v}}) \leq_{\mathtt{T}} [\![\mathtt{g}]\!](\mathtt{v}', \overline{\mathtt{v}})$ for any $\overline{\mathtt{v}}$;

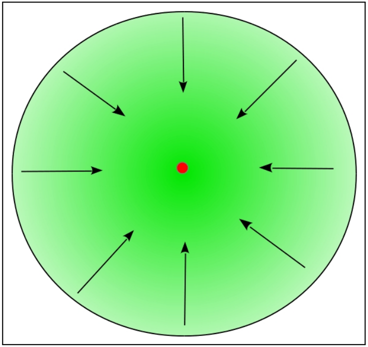
(iv) $\mathtt{g}$ is *progressive* in its first argument — $\mathtt{v} <_{\mathtt{T}} [\![\mathtt{g}]\!](\mathtt{v}, \overline{\mathtt{v}})$ (for $\mathtt{v} \neq \mathtt{top}(\mathtt{T})$).

Self-stabilisation result

A field expression whose spreading aggregation constructs feature only self-stabilising progressions is self-stabilising

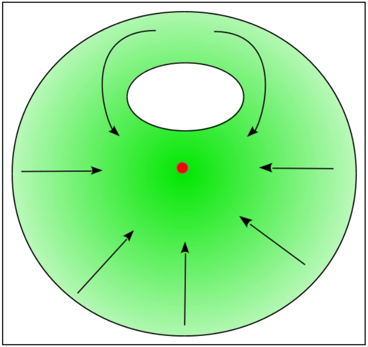
Distance-to

```
def int distanceto(int i) is { i : @ + #dist }
```



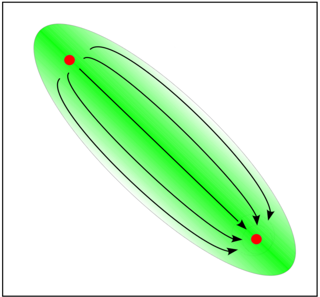
Distance-to cirumventing an obstacle

```
def int distanceto(int i) is { i : @ + #dist }
def int restrict(int i, bool b) is b ? i : INF
def int distobs(int i, bool b) is { i : restrict(@ + #dist, b) }
```



Channel

```
def int distanceto(int i) is { i : @ + #dist }
def int restrict(int i, bool b) is b ? i : INF
def int distobs(int i, bool b) is { i : restrict(@ + #dist, b) }
def <int,int> add_to_1st(<int,int> x, int y) is <1st(x)+ y, 2nd(x)>
def <int,int> broadcast(int i, int j) is { <i, j>: add_to_1st(@, #dist) }
def int dist(int i, int j) is broadcast(restrict(j,j==0),distanceto(i))
def bool path(int i, int j, int w) is
    distanceto(i) + distanceto(j) < dist(i,j) + w
def int channel(int i, int j, int w) is
    broadcast(distanceto(j),not path(i,j,w))
```



The proof

Structure of the proof

- Because of layering can reason inductively on expression structure
- The only non-trivial case is to prove that $\{\mathtt{v}_0 : \mathtt{g}(\mathtt{@}, \mathtt{v}_1, \ldots, \mathtt{v}_n)\}$ self-stabilises ($\mathtt{v}_i$ are immutable fields)

Key idea

Inductively on the size of the subnetwork S that already self-stabilised

- Base case: the node holding the minimum value of $\mathtt{v}_0$ self-stabilises immediately (at its first round)
- Inductive case: after a small transient, the minimum value outside S necessarily increases until $S \neq \oslash$, so there's surely another node that will self-stabilise:
  - either one reaching top(T),
  - or one in the neighbourhood of a node in S

  ... in both cases the result is independent of the initial state

Non-self-stabilising fields

```
;; previous definition
(def goss-max (F) (rep x F (max-hood (nbr x))))

;; non-self-stabilising
(goss-max (sense 1))

;; still not working
(def goss-max2 (F) (rep x F (max-hood (nbr x))))

;; where's the bug?
(def wrong-distance-to (SRC)
    (rep x (inf) (mux SRC 0 (min-hood (+ (nbr-range) (nbr x))))))
```

Conclusions

Other predictability studies (in the following)

- Eventual consistency, i.e., network density independence
- Universality of field calculus constructs

Open issues

- There are more self-stabilising fields than our condition can check
- The notion of self-stabilisation can be generalised
- What about correct dynamic fields?
- There's room for some more foundational work
- You do not like the language itself? Others are under study..

References I

[Barendregt, 1984] Barendregt, H. P. (1984).
The Lambda Calculus.
North Holland, revised edition.

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

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

[Dolev, 2000] Dolev, S. (2000).
Self-Stabilization.
MIT Press.

[Igarashi et al., 1999] Igarashi, A., Pierce, B., and Wadler, P. (1999).
Featherweight Java: A minimal core calculus for Java and GJ.

[Milner, 1999] Milner, R. (1999).
Communicating and Mobile Systems: The π-calculus.
Cambridge University Pres.

[MIT Proto, 2012] MIT Proto (Retrieved June 1st, 2012).
MIT Proto.
software available at http://proto.bbn.com/.

[Viroli and Damiani, 2014] Viroli, M. and Damiani, F. (2014).
A calculus of self-stabilising computational fields.
In eva Kühn and Pugliese, R., editors, *Coordination Languages and Models*, volume 8459 of *LNCS*, pages 163–178. Springer-Verlag.
Proceedings of the 16th Conference on Coordination Models and Languages (Coordination 2014), Berlin (Germany), 3-5 June.

[Viroli et al., 2013] Viroli, M., Damiani, F., and Beal, J. (2013).
A calculus of computational fields.
In Canal, C. and Villari, M., editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg.

[Zambonelli et al., 2011] Zambonelli, F., Castelli, G., Ferrari, L., Mamei, M., Rosi, A., Serugendo, G. D. M., Risoldi, M., Tchao, A.-E., Dobson, S., Stevenson, G., Ye, J., Nardini, E., Omicini, A., Montagna, S., Viroli, M., Ferscha, A., Maschek, S., and Wally, B. (2011).
Self-aware pervasive service ecosystems.
Procedia CS, 7.