Università degli Studi di Torino

Computer Science Department

Master Thesis

A DOMAIN SPECIFIC LANGUAGE FOR AGGREGATE PROGRAMMING

Supervisor
Prof. Ferruccio Damiani

Co-supervisor
Dr. Audrito Giorgio
Dr. Pianini Danilo

Opponent
Dr. Gianluca Torta

Candidate
Lorenzo Testa

October 2020
a.y. 2019 / 2020

A Domain Specific Language for Aggregate Programming

Lorenzo Testa

20th October 2020

Contents

Chapter 1

Introduction

Chapter 2

Background and state of the art

Chapter 3

Aggregate Programming

Aggregate programming [3] is an emerging framework and paradigm for the development of Collective Adaptive Systems. It is based on a layered architecture with which the developers can describe the system as an "aggregate" of heterogeneous devices, abstracting from the details of coordination and comunication and instead focusing on the collective behavior. The foundation of the Aggregate Programming is the *field calculus* [13], a functional programming model that unifies local and aggregate semantic.

3.1 Field Calculus

The *field calculus* is a programming model based on the notion of *computational fields* [9] (or simply *field*). A *field* is a distributed map from devices to computation objects across time. Therefore the field calculus describes how to build those distributed structure and reusable blocks of computation from fields to fields.

The computational model of the field calculus is based on a network of devices that executes a common program in asyncronous rounds. These devices comunicate with neighbour devices following a dynamic (physical or logical) proximity relation. From the local point of view of a single device every round of execution is composed by the following steps: (i) all the information from sensors and the device memory are collected, (ii) from the most recent messages from neighbouring devices a *neighbouring field* is formed, (iii) the program is executed with the collected information, (iv) the results of the computation are stored in the device memory and shared to the neighbouring devices as a message. A device δ is said to "fire" when it runs a round of execution.

From the aggregate point of view the whole computation can be seen as a space-time data structure, called *field evolution* Φ. Every execution is represented by a point in space-time called an *event* ϵ, Φ is then a map from events to computations

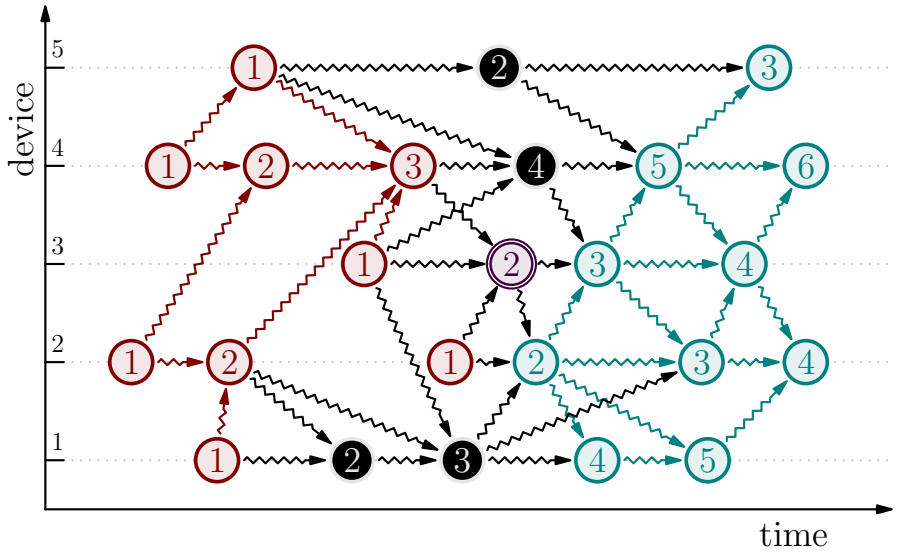

Figure 3.1: Example of a space-time event structure from [1], comprising events (circles), neighbour relations (arrows) and devices (ordinate axis). With respect to the doubly-circled event, the red events are its causal past, the cyan its causal future and the black ones are concurrent.

values. As described in [2] the causal relationship between events can be formalized by an *event structure*.

An *event structure* **E** is a countable set of events E togheter with a neighbouring relation $\rightsquigarrow \subseteq E \times E$ and a causality relation $< \subseteq E \times E$, such that the transitive closure of $\rightsquigarrow$ forms the irreflexive partial order $<$ and the set $\{\epsilon' \in E | \epsilon' < \epsilon\}$ is finite for all ϵ (i.e., $<$ is locally finite). Every $\rightsquigarrow$ relation represent a message sent from the head neighbour to the tail neighbour with the results of the head computation.

Figure 3.1 shows an example of an event structure, showing the relations among events.

The field calculus is a tiny functional language based on a set of abstract operators for the field computations. In this thesis only an higher-order extension of the field calculus, called *higher-order field calculus (HFC)* [13], will be considered. HFC extends the field calculus by treating function as first-class values and will be simply refered to as *field calculus* from now on.

The set of abstract operators is provided in figure 3.2. Following the notation of [7] the overbar denotes a sequence, for example $\overline{\mathsf{e}}$ denotes a (possible empty) sequence of expressions $\mathsf{e}_1, \mathsf{e}_2, \ldots, \mathsf{e}_n$.

A program is then a sequence of function definition followed by a main expression e, which defines the behavior of the aggregate.

P	::=	$\overline{\texttt{F}}$ e	program
F	::=	def d($\overline{\texttt{x}}$) {e}	function declaration
e	::=	x | v | e($\overline{\texttt{e}}$) | if(e){e}else{e} | nbr{e} | rep(e){(x)=>e} | share(e){(x)=>e}	expression
v	::=	ℓ | ϕ	value
ℓ	::=	c($\overline{\ell}$) | f	local value
ϕ	::=	$\overline{\delta} \mapsto \overline{\ell}$	neighbouring field value
f	::=	d | b | ($\overline{\texttt{x}}$) $\overset{\tau}{\texttt{=>}}$ e	function value

Figure 3.2: Abstract syntax of the field calculus from [13, 1]

A function declaration defines a function named d with a sequence of variable names $\overline{\texttt{x}}$ and a body of the function consisting in an expression e. The defined functions can be recursive.

An expression can be:

- a variable `x` referring a function parameter
- a function call $\texttt{e}(\overline{\texttt{e}})$, where `e` evaluates to a field of functions `f`, $\overline{\texttt{e}}$ are the function arguments and evaluates to the function application
- a branching expression $\texttt{if}(\texttt{e}_0)\{\texttt{e}_1\}\{\texttt{e}_2\}$, also called *domain restriction expression*, its a lazy evaluated expression that divides the computation in two branches: the devices for which $\texttt{e}_0$ evaluates to `True` computes $\texttt{e}_1$, the devices for which $\texttt{e}_0$ evaluates to `False` coputes $\texttt{e}_2$
- an `nbr`-*expression*, also called *neightbouring field construction*, `nbr{e}` which evaluates to a field from neighbouring devices (including the execution device) to their most recent evaluation of the expression `e`
- a `rep`-*expression*, also called *time evolution expression*, $\texttt{rep}(\texttt{e}_0)\{(\texttt{x})\texttt{=>e}\}$, which at each round evaluates to the application to the function of the result of the previous round, using the *initialization expression* $\texttt{e}_0$ in the first round
- a `share`-*expression* $\texttt{share}(\texttt{e}_1)\{(\texttt{x})\texttt{=>e}_2\}$, which each round evaluates to the application of the function with the argument `x` taking the value of a field with the last computed values $\texttt{e}_2$ by each neighbouring device, using $\texttt{e}_1$ as the initial value for the executing device.

A value can be either a *neighbouring field* ϕ or a *local value* ℓ. Neighbouring field values doesn't appear in the source code but can only be computed dynamically, usually by built-in operators like `nbr`.

Local values can be either *data value* $\texttt{c}(\overline{\ell})$, in which `c` its a data constructor and $\overline{\ell}$ are local value arguments, or a function value `f`.

A *function value* `f` can be a built-in function `b`, a declared function `d` or an anonymous function value $(\overline{\texttt{x}})\ \overset{\tau}{\texttt{=>}}\ \texttt{e}$ where $\overline{\texttt{x}}$ are variable names for the formal parameter, `e` is the body of the function and τ is a `tag` identifying the function. τ doesn't appear in the source code but is uniquely determined by the function syntactical representation.

3.2 Field Calculus Semantic

The operational semantics of the syntax is shown in figure 3.3. The derive judgements are in the form $\delta; \Theta; \sigma \vdash \texttt{e} \Downarrow \theta$ which means that the expression `e` evaluates to the value-tree θ with respect to the value-tree environment Θ, the device δ and the sensor state σ. A *value-environment* Θ is map from device identifies δ to value-trees. A *value-tree* θ is an ordered tree of values tracking the results of all the computed subexpressions. The evaluation rules are expressed recursively by evaluating the subexpressions with respect to a new value environment obtained by the subtrees (when present) of the current value-tree environment Θ, this process is called *alignment*. σ is a data structure containing information about the device sensors that will be used by the built-in functions.

The auxiliary function ρ extract the root value of a value-tree, while π extracts a subtree from a value-tree. The functions *name*, *args* and *body* extract respectively the name, formal parameters and body of a function.

Rules [E-LOC] and [E-FLD] define the evaluation of local values and neighbouring field values. Both produce a value-tree with no subtrees, but in case of neighbouring fields the domain of the field is restrict to the aligned devices.

Rule [E-B-APP] and [E-D-APP] model the application of built-in and user-defined (or anonymous) functions. In the first case the root of the value-tree is computed by a function $(\!|\texttt{b}|\!)_{\delta}^{\pi^{\texttt{b}}(\Theta),\sigma}$ different for each built-in function `b`. In the second case the root is computed by execution the body of the function `f`, the resulting value-tree also has one additional subtree containing the value-tree resulting from the execution from the body. This is necessary for the alignment of the environment during the execution.

Rule [E-NBR] models the evaluation of `nbr`-expression, which extracts from the value-tree environment the neighbouring values to build a neighbouring field as the root result. In the resulting field the value associated with the executing device is updated by the new result of the execution of `e`.

Rule [E-REP] models the evaluation of `rep`-expression, which extract from the value-tree environment the root of the last computed tree in order to replace `x` in the new evaluation of $\texttt{e}_2$.

Rule [E-IF] models a branching expression by computing and aligning on only one subtree according to the evaluation of `e`. This rule is actually not necessary since every `if` expression can be rewritten by using the built-in operator $\texttt{mux}(\texttt{e}, \texttt{e}_1, \texttt{e}_2)$ which eagerly evaluates both $\texttt{e}_1$ and $\texttt{e}_2$ and returns the result of one of them according to the truth value computed by `e`. Every $\texttt{if}(\texttt{e})\{\texttt{e}_1\}\{\texttt{e}_2\}$ can be rewritten then as $\texttt{mux}(\texttt{e}, ()\texttt{=>}\texttt{e}_1, ()\texttt{=>}\texttt{e}_2)()$.

Rule [E-SHARE] models a `share`-expression, which collects the neighbouring values of the last computations of the expression to form the field ϕ. In case there is not a value for the device δ the root of the evaluation of $\texttt{e}_1$ is used. Then ϕ is substituted to `x` in the evaluation of $\texttt{e}_2$.

3.3 Aggregate Programming Layers

TODO add figure

From the field calculus the aggregate programming framework is built as a series of layers, visibles in figure TODO. The *resilient coordination operators layer* defines using the operators of the field calculus a series of functions that hide the complexity of the basic operators and restric the language to a self-stabilising fragment of the field calculus [12]. Then over this operators aggregate programming libraries provides reusable and flexible high level developer APIs, e.g. function for broadcasting values, to computed distances among devices, etc. The application code is then developed on the reusable blocks provided by the libraries.

The resilient coordination layer defines in particular the following three operators:

- *Block* $\texttt{G}(\texttt{source}, \texttt{initial}, \texttt{metric}, \texttt{accumulate})$, a spreading operator for distance measurement and broadcast of values. It computes the shortest-path from a `source` (field with value `True` for sources) accourting to a `metric` (function mapping neightbours to distance) and propagate values up the gradient starting with the value of `initial` and accumulating with the binary function `accumulate`

- *Block* $\texttt{C}(\texttt{potential}, \texttt{local}, \texttt{null}, \texttt{accumulate})$, an operator that accumulates values with the binary function `accumulate` down to the `source` following the `potential` field. `null` provides the idempotent value for the accumulation function, `local` is accumulated with any values from neighbours at higher potential

- *Block* `T(initial, zero, decay)`, a flexible countdown operator starting from `initial` to `zero` decreasing by the `decay` function.

Those operators are able to cover many of the common patterns and define a self-stabilising fragment of the field calculus. A computation is self-stabilizing if from any state, without changes of any environment, the computation reaches after a certain number of round a correct final result.

Value-trees and value-tree environments:

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

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

Auxiliary functions:

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

$\pi_i(\mathtt{v}\langle\theta_1, \ldots, \theta_n\rangle) = \theta_i \quad \text{if } 1 \leq i \leq n$

$\pi^{\mathtt{f}}(\mathtt{v}\langle\theta_1, \ldots, \theta_{n+1}\rangle) = \theta_{n+1} \quad \text{if } \mathtt{f} \text{ is a built-in function and } \rho(\theta_{n+1}) = \mathtt{f}$

$\pi^{\mathtt{f}}(\mathtt{v}\langle\theta_1, \ldots, \theta_{n+2}\rangle) = \theta_{n+2} \quad \text{if } \mathtt{f} \text{ is a non-built-in function and } \mathit{name}(\rho(\theta_{n+1})) = \mathit{name}(\mathtt{f})$

$\pi^{\mathtt{f}}(\theta) = \bullet \quad \text{otherwise}$

For $\mathit{aux} \in \rho, \pi_i, \pi^{\mathtt{f}}$:

$$\begin{cases} \mathit{aux}(\delta \mapsto \theta, \Theta) = \delta \mapsto \mathit{aux}(\theta), \mathit{aux}(\Theta) & \text{if } \mathit{aux}(\theta) \neq \bullet \\ \mathit{aux}(\delta \mapsto \theta, \Theta) = \mathit{aux}(\Theta) & \text{if } \mathit{aux}(\theta) = \bullet \\ \mathit{aux}(\bullet) = \bullet \end{cases}$$

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

$\mathit{name}((\overline{\mathtt{x}}) \overset{\tau}{\texttt{=>}} \mathtt{e}) = \tau \qquad \mathit{args}((\overline{\mathtt{x}}) \overset{\tau}{\texttt{=>}} \mathtt{e}) = \overline{\mathtt{x}} \qquad \mathit{body}((\overline{\mathtt{x}}) \overset{\tau}{\texttt{=>}} \mathtt{e}) = \mathtt{e}$

$\phi_0[\phi_1] = \phi_2 \text{ where } \phi_2(\delta) = \begin{cases} \phi_1(\delta) & \text{if } \delta \in \mathcal{D}_{\phi_1} \\ \phi_0(\delta) & \text{otherwise} \end{cases}$

Syntactic shorthands:

$\delta; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta} \text{ where } |\overline{\mathtt{e}}| = n \text{ for } \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 \cdots \delta; \pi_n(\Theta); \sigma \vdash \mathtt{e}_n \Downarrow \theta_n$

$\rho(\overline{\theta}) \text{ where } |\overline{\theta}| = n \text{ for } \rho(\theta_1), \ldots, \rho(\theta_n)$

$\overline{\mathtt{x}} := \rho(\overline{\theta}) \text{ where } |\overline{\mathtt{x}}| = n \text{ for } \mathtt{x}_1 := \rho(\theta_1) \ \ldots \ \mathtt{x}_n := \rho(\theta_n)$

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

$$\frac{}{\delta; \Theta; \sigma \vdash \ell \Downarrow \ell\langle\rangle}\ [\text{E-LOC}] \qquad \frac{\phi' = \phi|_{\mathcal{D}_\Theta \cup \{\delta\}}}{\delta; \Theta; \sigma \vdash \phi \Downarrow \phi'\langle\rangle}\ [\text{E-FLD}]$$

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

$$\frac{\begin{array}{c}\delta; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}}, \mathtt{e} \Downarrow \overline{\theta}, \theta \qquad \mathtt{f} = \rho(\theta) \text{is not a built-in} \\ \delta; \pi^{\mathtt{f}}(\Theta); \sigma \vdash \ \mathit{body}(\mathtt{f})[\mathit{args}(\mathtt{f}) := \rho(\overline{\theta})] \Downarrow \theta'\end{array}}{\delta; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \Downarrow \rho(\theta')\langle\overline{\theta}, \theta, \theta'\rangle}\ [\text{E-D-APP}]$$

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

$$\frac{\begin{array}{l}\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 \\ \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2[\mathtt{x} := \ell_0] \Downarrow \theta_2\end{array} \qquad \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathcal{D}_\Theta \\ \rho(\theta_1) & \text{otherwise} \end{cases}}{\delta; \Theta; \sigma \vdash \mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\ \texttt{=>}\ \mathtt{e}_2\} \Downarrow \rho(\theta_2)\langle\theta_1, \theta_2\rangle}\ [\text{E-REP}]$$

$$\frac{\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta_1 \qquad \ell_0, \ell_1 = \begin{cases} \pi_1(\Theta), \mathtt{e}_1 & \text{if } \rho(\theta_1) = \mathtt{True} \\ \pi_2(\Theta), \mathtt{e}_2 & \text{if } \rho(\theta_1) = \mathtt{False} \end{cases} \qquad \delta; \ell_0; \sigma \vdash \ell_1 \Downarrow \theta}{\delta; \Theta; \sigma \vdash \mathtt{if(e)\{e_1\}\{e_2\}} \Downarrow \rho(\theta)\langle\theta_1, \theta\rangle}\ [\text{E-IF}]$$

$$\frac{\begin{array}{l}\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 \qquad \phi' = \rho(\pi_2(\Theta)) \qquad \phi = (\delta \mapsto \rho(\theta_1))[\phi'] \\ \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2[\mathtt{x} := \phi] \Downarrow \theta_2\end{array}}{\delta; \Theta; \sigma \vdash \mathtt{share}(\mathtt{e}_1)\{(\mathtt{x})\ \texttt{=>}\ \mathtt{e}_2\} \Downarrow \rho(\theta_2)\langle\theta_1, \theta_2\rangle}\ [\text{E-SHARE}]$$

Figure 3.3: Big-step operational semantics adapted from [13].

Chapter 4

Protelis

Protelis [11] is a Domain Specific Language providing a practical implementation of the aggregate programming paradigm. It runs on the Java Virtual Machine (JVM) and provides full interoperability with the Java type-system and API. The text Protelis programs are translated into a valid representation of the higher order field calculus semantics, then this representation is executed at regular intervals by the Protelis interpreter. Protelis abstracts over the device capabilities and communication system, allowing to use it for both simulations (like the Alchemist simulator [10]) and real world application. Protelis also provides a rich standard library for the application developers.

Figure 4.1 shows the abstract syntax of the Protelis syntax. A Protelis program is composed by a sequence of Java imports, followed by a sequence of function declarations and a sequence of statements composing the main code. Each import specifies the package (if any) and the method name. Each function definition declares a function named `f` with a sequence of variables named $\overline{\texttt{x}}$ and the body composed by a sequence of statements $\overline{\texttt{s}}$. The result of a sequence of statements is always considered the value of the last statement of the sequence.

A statement can be an expressions `e`, a local variable declaration in the form `let x = e` where `x` is the new variable name and `e` is the expression computing the initial value of the variable, or a re-assignment of a new value to a variable in the form `x = e` where `x` is the name of an existing variable and `e` is the expression computing the new value.

A value can be a variable name `x`, a literal value `l` (Boolean, numerical, string), a tuple of values in the form $[\overline{\texttt{w}}]$, a function name `f` or a lambda (i.e. an anonymous function) in the form $(\overline{\texttt{x}})\texttt{->\{}\ \overline{\texttt{s}};\texttt{\}}$ where $\overline{\texttt{x}}$ are the arguments and the body is a sequence of statements $\overline{\texttt{s}}$.

Finally an expression can be:

- a value `w`

P	::=	$\overline{\texttt{I}}\ \overline{\texttt{F}}\ \overline{\texttt{s}}$;	Program
I	::=	`import m` | `import m.*`	Java import
F	::=	`def f`$(\overline{\texttt{x}})$ $\{\overline{\texttt{s}}\}$	Function definition
s	::=	`e` | `let x = e` | `x = e`	Statement
w	::=	`x` | `l` | $[\overline{\texttt{w}}]$ | `f` | $(\overline{\texttt{x}})$`->{ `$\overline{\texttt{s}}$`; }`	Variable/Value
e	::=	`w` |	Expression
		`b`$(\overline{\texttt{e}})$ | `f`$(\overline{\texttt{e}})$ | `e.apply`$(\overline{\texttt{e}})$ |	Fun/Op Calls
		`e.m`$(\overline{\texttt{e}})$ | `#a`$(\overline{\texttt{e}})$ |	Method Calls
		`rep(x<-w){`$\overline{\texttt{s}}$`; }` |	Persistent state
		`if(e){`$\overline{\texttt{s}}$`; }else{`$\overline{\texttt{s}}'$`; }` |	Exclusive branch
		`mux(e){`$\overline{\texttt{s}}$`; }else{`$\overline{\texttt{s}}'$`; }` |	Inclusive branch
		`nbr{`$\overline{\texttt{s}}'$`; }`	Neighbourhood values

Figure 4.1: Abstract syntax of the Protelis language from [11]

- a function call with the arguments $\overline{\texttt{e}}$, the function can be either a built-in function `b`[1], a user defined function `f` or the application of the argument to a lambda or function name resulting from the evaluation of `e`
- a Java method call with the arguments $\overline{\texttt{e}}$, it can be called either a method `m` on an object computed by `e` or a static method via an alias `#a`. The aliases are created automatically from the imports.
- a `rep`-construct `rep(x<-w)`$\{\overline{\texttt{s}}; \}$, equivalent to the field calculus `rep`, declaring a variable `x` inizialized at `w` and updated each round with the result of the execution of the body $\overline{\texttt{s}}$
- an `if`-construct `if(e)`$\{\overline{\texttt{s}}; \}$`else`$\{\overline{\texttt{s}}'; \}$, equivalent to the field calculus `if`, execution only $\overline{\texttt{s}}$ or $\overline{\texttt{s}}'$ according to the branching condition computed by `e`
- a `mux`-construct `mux(e)`$\{\overline{\texttt{s}}; \}$`else`$\{\overline{\texttt{s}}'; \}$, computing both branches $\overline{\texttt{s}}$ and $\overline{\texttt{s}}'$ and return the value of one of them accourding to the branching condition computed by `e`
- a `nbr`-construct `nbr`$\{\overline{\texttt{s}}'; \}$, equivalent to the field calculus `nbr`, returning a field from all neighbors to their last value from computing $\overline{\texttt{s}}$.

[1] some built-in function can be called with an infix-style but the syntax has been omitted for simplicity

```
def count() { rep(x <- 0){ x + 1 } }
def maxh(field) { maxHood(nbr{field}) }
def distanceTo(source) {
  rep(d <- Infinity) {
    mux (source) { 0 }
    else { minHood(nbr{d} + nbrRange) }
  }
}
def distanceToWithObstacle(source, obstacle) {
  if (obstacle) { Infinity } else { distanceTo(source) }
}
```

Figure 4.2: Examples of Protelis code from [11]

Figure 4.2 shows an example of Protelis code. The function `count` yields the number of round that have been executed. The function `maxh` yield a the maximum value of `field` across all neighbours using the built-in function `maxHood`, which takes a field as argument an returns the maximum value. Its important to note that the only way to extract a regular value from a field in Protelis is to use one of the many built-in "hood" functions. The function `distanceTo` computes the distance from the device to the nearest device where `source` holds `True`, each device starts with an infinite distance and each round returns zero if it is a source otherwise returns the minimum of the distances shared by the neighbours summed to the distance to that neighbour. It uses the built-in functions `minHood` and `nbrRange`, the former returning the minimum value in the field and the latter returning a field of the distances to each neightbour. The last function `distanceToWithObstacle` splits the network in two sub-regions, the normal nodes simply computes the function `distanceTo` while the nodes where obstacle holds `True` don't partecipate in the algorithm and return an infinite distance.

Chapter 5

Scafi

Scafi [5, 4] is a library defining a Domain Specific Language that integrates the aggregate paradigm into the Scala programming language [6]. Like Protelis it runs on the Java Virtual Machine but instead of defining its own language it take advantage of the Scala language and its expressive type system. Scafi unlike the others aggregate programming implementation doesn't have an explicit representation of fields but instead has a notion of "computation against a neighbour", i.e. a computation depending on the evaluation of the same expression by an alligned neighbour. This difference aims to provide an embedding of field computations into Scala more close to the host language. It also provides an integration with Akka [8], a Scala industry-ready actor framework for scalable and resilient message-driven applications.

Scafi is an *internal DSL* which means it provides its API on top of the host programming language, unlike Protelis which is an *external DSL*, offering a standalone language. This means that developers doesn't need to learn a new language and the Scafi mantainers don't need to develop editing tools reusing instead the ones for the Scala language. There is also experimental evidence that Protelis programs require to delegate a significant part of the code to Java methods for code reuse or simplicity. This increases the complexity of development since the code in both languages must be keep coordinated.

A Scafi program is a function that extends an implementation of the trait `Constructs`, visible in figuera 5.1, which provides all the aggregate programming basic constructs.

The method `rep[A](init : =>A)(fun : (A)=>A) : A` implements a `rep`-expression of the field calculus, returning a value of type `A`. `init` is a lazy expression computing the beginning value. At each round the `fun` unary function is applied to compute the new returned value from the last value.

The method `fooldhood[A](init : =>A)(aggr : (A, A)=>A)](expr : =>A) : A` pro-

```
trait Constructs {
  // Key constructs
  def rep[A](init: => A)(fun: (A) => A): A
  def foldhood[A](init: => A)(aggr: (A, A) => A)(expr: => A): A
  def nbr[A](expr: => A): A
  def @@[A](b: => A): A

  // Abstract types
  type ID
  type LSNS, NSNS

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

Figure 5.1: Scafi constructs from [5]

vides a way to extract and combine information from neightbours. First the lazy expression `expr` is computed against each aligned neighbour, each computation producing a value of type `A`. Then the results are combined into a single value using the associative binary function `aggr` with his neutral element `init`.

The method `nbr[A](expr : =>A) : A` defines a neightbour dependent expression that return a value of type `A`. The result is computed by the lazy expression `expr` when evaluated against the local device. When evaluated against a neightbouring device it returns instead the most recent value of `expr` computed by that device.

The method `@@[A](b : =>A) : A` is a function that performs an alignment, allowing to perform branching in the execution. Inside the body `b` only the neightbours executing the same body are considered aligned. For example the `branch` method `branch[A](cond : Boolean)(th : =>A)(el : =>A) : A`, corresponding to the `if` in the field calculus, is defined as `mux(cond)(()=> @@{th})(()=> @@{el})()` where `mux` is an eagerly evaluated version of the Scala `if`.

Finally `mid` returns the unique identifier, of the abstract type `ID`, of the running device, `sense` reads from the local sensor named `name`, of the abstract type `LSNS`, its current value of type `A` and `nbrvar` reads from a neighbouring sensor named `name` (a sensor of the running device which value dependens on the neighbour against which is currently computing), of the abstract type `NSNS`, its current value of type `A`. The device temperature or GPS position are examples of local sensors, while the distance from a neighbour is an example of neighbouring sensor.

Figure 5.2 shows some example of Scafi programs. The first function `count` counts the number of rounds executed by the device. The function `countNeighbours`

```
def count(): Int = { rep(0){ x => x + 1} }
def countNeighbours(): Int = {
  foldhood(0)(_ + _)(1)
}
def maxh(field: Double): Double = {
  foldhood(Double.NegativeInfinity)(Math.max(_,_)){ nbr(field) }
}
```

Figure 5.2: Examples of Scafi programs

counts how many neighbours has the device, simply by adding one for each device. The function `maxh` returns the maximum value `field` with which is called acroos neighbours. This is accomplisced by calling `nbr` against each neighbour and combining the values by the Scala `Math.max` function.

Chapter 6

Differences between Scafi semantics and field calculus

Chapter 7

Kotlin

Chapter 8

KotAC

Chapter 9

Future developments

References

[1] G. Audrito, J. Beal, F. Damiani, D. Pianini, and M. Viroli. Field-based coordination with the share operator, 2019.

[2] G. Audrito, J. Beal, F. Damiani, and M. Viroli. *Space-Time Universality of Field Calculus*, pages 1–20. 01 2018.

[3] J. Beal, D. Pianini, and M. Viroli. Aggregate programming for the internet of things. *Computer*, 48:22–30, 09 2015.

[4] R. Casadei. *Engineering Self-Adaptive Collective Processes for Cyber-Physical Ecosystems*. PhD thesis, alma, Aprile 2020.

[5] R. Casadei and M. Viroli. Towards aggregate programming in scala. In *First Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '16, New York, NY, USA, 2016. Association for Computing Machinery.

[6] EPFL. The scala programmign language. https://scala-lang.org/.

[7] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[8] Lightbend. Akka: build concurrent, distributed, and resilient message-driven applications for java and scala. https://akka.io/.

[9] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18, 07 2009.

[10] D. Pianini, S. Montagna, and M. Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7, 01 2013.

[11] D. Pianini, M. Viroli, and J. Beal. Protelis: Practical aggregate programming. *Proceedings of the ACM Symposium on Applied Computing*, pages 1846–1853, 01 2015.

[12] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini. Engineering resilient collective adaptive systems by self-stabilisation. 28(2), Mar. 2018.

[13] M. Viroli, G. Audrito, F. Damiani, D. Pianini, and J. Beal. A higher-order calculus of computational fields. *ACM Transactions on Computational Logic*, 20, 10 2016.