

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

1	Introduction	5
2	From distributed systems to Aggregate Programming	7
3	Aggregate Programming	9
3.1	Field Calculus	9
3.2	Aggregate Programming Layers	11
4	Protelis	13
5	Safi	17
6	Kotlin	19
7	Kotac	21
8	Future developments	23
	References	23

CHAPTER 1

Introduction

CHAPTER 2

From distributed systems to Aggregate Programming

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 communication 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 asynchronous rounds. These devices communicate 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

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

Figure 3.1: Abstract syntax of the field calculus from [13]

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

An *event structure* \mathbf{E} is a countable set of events E together 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 $\{\epsilon' \in E \mid \epsilon' < \epsilon\}$ 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.

TODO add figure.

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 referred to as *field calculus* from now on.

The set of abstract operators is provided in figure 3.1. Following the notation of [7] the overbar denotes a sequence, for example \bar{e} denotes a (possible empty) sequence of expressions e_1, e_2, \dots, e_n .

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

A function declaration defines a function named d with a sequence of variable names \bar{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
- an anonymous function $(\bar{x}) \stackrel{\tau}{\Rightarrow} e$, where \bar{x} are variable names for the formal

parameter, e is the body of the function and τ is a *tag* identifying the function. It doesn't appear in the source code but is uniquely determined by its syntactical representation

- a function call $e(\bar{e})$, where e evaluates to a field of functions f , \bar{e} are the function arguments and evaluates to the function application
- a branching expression $\text{if}(e_0)\{e_1\}\{e_2\}$, also called *domain restriction expression*, its a lazy evaluated expression that divides the computation in two branches: the devices for which e_0 evaluates to **True** computes e_1 , the devices for which e_0 evaluates to **False** computes e_2
- an *nbr-expression*, also called *neighbouring field construction*, $\text{nbr}\{e\}$ 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*, $\text{rep}(e_0)\{(x) \Rightarrow \{e\}\}$, which at each round evaluates to the application to the function of the result of the previous round, using the *initialization expression* e_0 in the first round.

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* $c(\bar{\ell})$, in which c its a data constructor and $\bar{\ell}$ 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 $(\bar{x}) \xRightarrow{\tau} e$.

An additional operator $\text{share}(e_0)\{(x) \Rightarrow e_1\}$ has been added in [1], which each round evaluates to the application to the function of the neighbouring field with the results of the previous round for each neighbouring device, using the expression e_0 in the first round as the value for the executing device. This new operator allows to write more efficient algorithms.

3.2 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 G*(**source**, **initial**, **metric**, **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) according to a **metric** (function mapping neighbours to distance) and propagate values up the gradient starting with the value of **initial** and accumulating with the binary function **accumulate**
- *Block C*(**potential**, **local**, **null**, **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.

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 \bar{x} and the body composed by a sequence of statements \bar{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 $[\bar{w}]$, a function name f or a lambda (i.e. an anonymous function) in the form $(\bar{x}) \rightarrow \{ \bar{s}; \}$ where \bar{x} are the arguments and the body is a sequence of statements \bar{s} .

Finally an expression can be:

- a value w

$P ::= \bar{I} \bar{F} \bar{s};$	Program
$I ::= \text{import } m \mid \text{import } m.*$	Java import
$F ::= \text{def } f(\bar{x}) \{ \bar{s} \}$	Function definition
$s ::= e \mid \text{let } x = e \mid x = e$	Statement
$w ::= x \mid 1 \mid [\bar{w}] \mid f \mid (\bar{x}) \rightarrow \{ \bar{s}; \}$	Variable/Value
$e ::= w \mid$	Expression
$\quad b(\bar{e}) \mid f(\bar{e}) \mid e.\text{apply}(\bar{e}) \mid$	Fun/Op Calls
$\quad e.m(\bar{e}) \mid \#a(\bar{e}) \mid$	Method Calls
$\quad \text{rep}(x \leftarrow w) \{ \bar{s}; \} \mid$	Persistent state
$\quad \text{if}(e) \{ \bar{s}; \} \text{else} \{ \bar{s}'; \} \mid$	Exclusive branch
$\quad \text{mux}(e) \{ \bar{s}; \} \text{else} \{ \bar{s}'; \} \mid$	Inclusive branch
$\quad \text{nbr} \{ \bar{s}'; \}$	Neighbourhood values

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

- a function call with the arguments \bar{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 \bar{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 $\text{rep}(x \leftarrow w) \{ \bar{s}; \}$, equivalent to the field calculus **rep**, declaring a variable x initialized at w and updated each round with the result of the execution of the body \bar{s}
- an **if**-construct $\text{if}(e) \{ \bar{s}; \} \text{else} \{ \bar{s}'; \}$, equivalent to the field calculus **if**, execution only \bar{s} or \bar{s}' according to the branching condition computed by e
- a **mux**-construct $\text{mux}(e) \{ \bar{s}; \} \text{else} \{ \bar{s}'; \}$, computing both branches \bar{s} and \bar{s}' and return the value of one of them according to the branching condition computed by e
- a **nbr**-construct $\text{nbr} \{ \bar{s}'; \}$, equivalent to the field calculus **nbr**, returning a field from all neighbors to their last value from computing \bar{s} .

¹some 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 and 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 neighbour. 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 participate in the algorithm and return an infinite distance.

CHAPTER 5

Safi

Safi [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. Safi 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. It also provides an integration with Akka [8], a Scala industry-ready actor framework for scalable and resilient message-driven applications.

CHAPTER 6

Kotlin

CHAPTER 7

Kotac

CHAPTER 8

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.