FLY: A Domain-Specific Language for Scientific Computing on FaaS

Gennaro Cordasco[1](✉), Matteo D'Auria[2], Alberto Negro[2], Vittorio Scarano[2], and Carmine Spagnuolo[2]

[1] Dipartimento di Psicologia, Università degli Studi della Campania, Caserta, Italy
gennaro.cordasco@unicampania.it
[2] Dipartimento di Informatica, Università degli Studi di Salerno, Fisciano, Italy
{matdauria,alberto,vitsca,cspagnuolo}@unisa.it

Abstract. Cloud Computing is widely recognized as distributed computing paradigm for the next generation of dynamically scalable applications. Recently a novel service model, called Function-as-a-Service (FaaS), has been proposed, that enables users to exploit the computational power of cloud infrastructures, without the need to configure and manage complex computations systems. FaaS paradigm represents an opportunity to easily develop and execute extreme-scale applications as it allows fine-grain decomposition of the application with a much more efficient scheduling on cloud provider infrastructure.

We introduce FLY, a domain-specific language for designing, deploying and executing scientific computing applications by exploiting the FaaS service model on different cloud infrastructures. In this paper, we present the design and the language definition of FLY on several computing (local and FaaS) back-ends: Symmetric multiprocessing (SMP), Amazon AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, and IBM Bluemix/Apache OpenWhisk. We also present the first FLY source-to-source compiler, publicly available on GitHub, which supports SMP and AWS back-ends.

Keywords: Domain-Specific Languages · Scientific computing · Parallel computing · Distributed computing · Serverless computing · Functions as a Service (FaaS)

1 Introduction

Cloud computing [2] is widely recognized as distributed computing paradigm for the next generation of dynamically scalable applications. Since the dawn of the practice of the cloud, many service models are competing to become the leading model of cloud infrastructures. Nowadays, Cloud computing is undergoing a service-model shift, moving the computation on the *Serverless computing model*, superseding the popular service-models as *Infrastructure-as-a-Service* (IaaS), *Platform-as-a-Service* (PaaS) and *Software-as-a-Service (SaaS)*. Serverless computing model is a novel paradigm for deployment of cloud applications, in which code snippets are executed over the cloud infrastructure without having to manage or configure the machines running the code.

Serverless computing architecture is the natural evolution of *microservices architecture* [3], in which the developers do not have to mind about the configuration and management of the servers executing the back-end of their applications. Cloud applications based on serverless computing are event-triggered: programmer-defined events rule the independent execution of modular pieces of code on the cloud environment. This novel service model, named Function-as-a-Service (FaaS), was first introduced and made available to the world by hook.io in late 2014 and was shortly followed by AWS Lambda, Google Cloud Functions, Microsoft Azure Functions and many others.

FaaS can be seen as a finegrained computing partitioning of a cloud applications, which enables to scale according to the provider capacity. FaaS has been designed for easily build and deploy scalable applications that are business-oriented such as Mobile and Internet of Things (IoT) Back-end, Real-time File/Stream Processing, Web Applications as well as service oriented applications.

In this work we take a novel approach by exploiting FaaS cloud service-model to develop scalable computing-intensive applications for scientific and data science. In fact, since the very beginning of Cloud, it was clear how the paradigm represented an opportunity to easily develop and execute extreme-scale applications maintaining their costs extremely low compared to High Performance Computing solutions, as shown by the experiments in [4]. On the other hand, although cloud providers offer solutions with a high level of scalability, very often the migration of a scientific application on IaaS or PaaS represents a humongous and complex task, which can conceal serious cost considerations, thereby often preventing scientific application developers to fully exploit the scalability and cost effectiveness of cloud computing in their own application domain. This work aims at reconciling Cloud and High Performance Computing by providing an efficient, effective and price-aware tool for the development of scalable scientific computing application on several FaaS environments through the design and implementation of a *domain-specific language* (DSL) named FLY. FLY is *efficient* because it enables to exploit the computing capabilities of different cloud providers at once, in a single application, and, then, the most efficient solutions can be merged together. FLY is *effective* because it consists of a user-friendly programming language that frees the programmer from the management and configuration of several complex computation systems. Finally, FLY is *price-aware* because the programmer becomes conscious of the maximum computing costs, based on the prices provided by various cloud providers. In this way, the programmer also has the possibility to choose the service that provides the best value for money, based on the characteristics of the computation that is going to perform.

1.1 The Motivations for a Parallel Language for FaaS

Cloud infrastructure provides several services in an accessible fashion through web endpoints, and/or APIs. Designing and developing scientific applications typically does not require general purposes services (for instance access to database or providing web pages), but it requires ad-hoc coding that implements algorithms, which solve specific problems.

Scientific computing applications are commonly developed using general-purposes languages or parallel languages/frameworks such as C, Java, Python, Fortran, Julia,

Table 1. Cloud Computing infrastructures API and FaaS programming languages fragmentation.

Cloud infrastructure	FaaS service	API languages	FaaS languages	Pricing and limitations
Amazon Web Services[b]	AWS Lambda function	Java, .NET, Node.js, PHP, Python, Ruby, Go, C++, REST	JavaScript, Java, Python, Go, C#	1 M functions and 400.000 GB/s of execution time free per month The execution time of a single function is limited at 300 s
Microsoft Azure[b]	Azure function	.NET, Java, Python, Go, Node.js, REST	C#, F#, JavaScript, Java	1 M functions and 400.000 GB/s of execution time free per month The execution time of a single function is limited at 300 s
Google[b]	Google function	REST, RPC	JavaScript	2 M functions, 1 M seconds of execution and 5 GB of network traffic free per month[a] The execution time of a single function is limited at 540 s
IBM Bluemix/Apache OpenWhisk[b]	Action	REST	JavaScript, Python, Java, PHP, Swift, Docker and native binaries, Go	5 M of functions and 400.000 GB/s of execution time free per month[b] The execution time of a single function is limited at 600 s
Fission[b]	Fission function	REST	C#, Go, JavaScript, PHP, Python	
Fn Project[b]	Fn function	REST	Java, Go,Ruby, Python, PHP, JavaScript	
Kubeless[b]	Kubeless function	REST	Python, JavaScript, Ruby, PHP, Go, .NET, Ballerina	

[a] [1] Amazon AWS Lambda pricing. [2] Microsoft Azure Function pricing.
[3] Google Function pricing. [4] IBM Bluemix pricing.
[b] *Amazon AWS Lambda*, aws.amazon.com/lambda.
Microsoft Azure Functions, azure.microsoft.com/services/functions.
Google Cloud Functions, cloud.google.com/functions.
IBM Bluemix, www.ibm.com/cloud-computing/bluemix.
Apache OpenWhisk, openwhisk.apache.org.
Fission, docs.fission.io.
Fn Project, fnproject.io.
The Kubernetes Native Serverless Framework, kubeless.io.

Limbo, Chapel, MPI, Swift and many others (see Sect. 3 for more details). Moreover, scientific computing problems are typically computing-intensive and requires the computational power of a distributed system (clusters or HPC). Since 2017, Amazon Inc. company provides, in their IaaS offer, machines with high number of virtual processors and memory, which enables users to execute applications on a high performance cluster “de facto”. According to the IaaS model, in such cases, the user is responsible for deploying and managing of such virtual clusters.

Although many cloud computing companies are recently providing MapReduce [5] programming paradigm as a cluster of machines running MapReduce compliant framework such as Apache Hadoop (e.g., AWS’s *Elastic Map Reduce*), many computing-intensive problems do not fit well the MapReduce paradigm.

FLY also addresses another issue about the nature and prices of the services offered by Cloud computing providers. In fact, in some cases, it would be extremely convenient, either in terms of efficiency or cost, to be able to develop cloud scientific applications exploiting different services coming from different providers. Our result, then, enables a scientific application designer to write computing-intensive applications that can scale-up among different computing providers at the lowest costs, selecting the services that best fit the requirements of the considered problem.

2 Preliminaries

This section presents and discusses the research and state-of-the-art for the cloud computing service-models domain as well as a short introduction to domain specific languages.

2.1 Cloud Computing Service-Models

Cloud computing enables companies to use computing resources as a service (like electricity) rather than having to buy, set-up and maintain computing infrastructures in house. Several cloud computing service-models [6] has been proposed during the last two decades. Three models are mainly used by cloud providers:

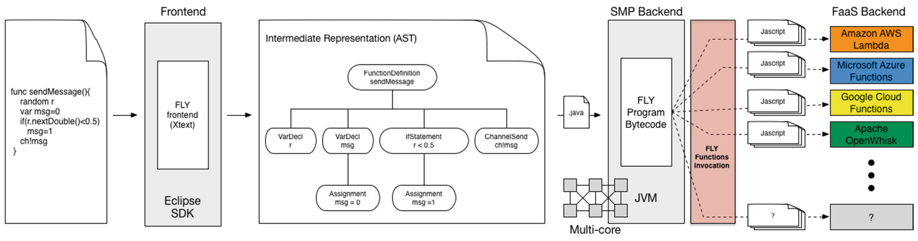
- *Software-as-a-Service (SaaS)*, when applications are hosted by a cloud providers and made available on the web.
- *Platform-as-a-Service (PaaS)*, which is a paradigm for delivering applications frameworks on the Internet without downloading or installing it.
- *Infrastructure-as-a-Service (IaaS)*, which can be seen as the outsourcing of computing power required by the customers. This involves disk space, hardware, and networking components.

At a first sight, Cloud service models look promising for the Scientific Computing community, as they may take advantage of the adoption of cloud computing, in their compute-intensive applications and workflows, in each of the service models described above. It is possible, for example, to use IaaS for executing application on high performance machine or huge clusters, or a cloud computing provider can offer either PaaS or SaaS, dedicated domain specific services for scientific and data analysis purposes (like machine-learning or data-mining services, MapReduce frameworks, etc.). But the scenario does not come without effort and costs, as, for example, the developers still need to manage (complex) virtual machines (IaaS), or configure the services (PaaS and SaaS). Moreover, the scalability of these systems depends on the configuration adopted and the overall performances and costs saving are strictly dependent on the fluency and skills of the developers in the Cloud Computing realm.

Serverless computing service-model (or Function-as-a-Service, Faas) [7–9] answers to the needs of new scalable price-effective cloud applications, by providing an easy framework for deploying extremely scalable, functionally partitioned applications. FaaS enables developers to run their back-end applications on complex computing systems, without a thorough knowledge of the management and configuration of such systems. Indeed, using FaaS, the user is able to execute independent piece of code (functions), written in different languages, over the cloud infrastructure, without taking care about which is and what kind of configuration has the server running the code. FaaS service-model architecture is event-triggered, which means that developers must deploy the functions on the cloud infrastructure, and those functions are executed in response to events generated on the cloud infrastructure (e.g., insert a new record in a database, send a message on a queue, etc.). Table 1 shows some of the most popular Cloud Computing infrastructures (open-source and private companies) that provide the FaaS service-model. Our proposal is guided by the vision to adopt this service-model in a different context, that is for computing-intensive applications.

2.2 Domain-Specific Languages

Domain-Specific Languages (DSLs) are designed to provide a notation tailored toward an application domain that is based only on the concepts and features that are relevant for the domain. DSLs enable solutions to be expressed at the same level of abstraction of the problem domain and can be of significant help in shifting the development of business information systems from software developers to a larger group of domain-experts who, despite having less technical expertise, have deeper knowledge of the domain and, therefore, if an easy-to-use, tailored tool is provided, can be much more effective. Furthermore, DSLs are much easier to learn, given their limited scope. It must be said that DSLs have specific design goals that contrast with those of general-purpose languages: DSLs are much more expressive in their domain and should exhibit minimal redundancy. Examples of DSL include SQL [10] (for relational database query), HTML [11] (for website definition), *R* [13] (for statistics), OpenABL [12] (for simulation).



```
func sendMessage(){
   random r
   var msg=0
   if(r.nextDouble()<0.5)
      msg=1
   ch!msg
}
```

Fig. 1. FLY compilation workflow.

3 Related Work

Parallel and distributed languages have been actively investigated for decades [15]. Here we describe several languages and frameworks that are suitable for developing scalable applications in the scientific computing (SC) research area.

General-Purpose Languages. *Fortran* is a programming language designed for numeric computation and scientific computing. It is widely used in scientific fields (such as numerical weather prediction, computational dynamics and physics). Programmers are moving toward modern programming languages like *Python* [17] and *Julia* [18]

Parallel Languages. *Limbo* [19] is a programming language intended for applications running distributed systems on small computers. *Chapel* [20] is a programming language designed for productive parallel computing on large-scale systems. Its design and implementation have been undertaken with portability in mind, enabling Chapel to run on different environments. *Cilk* [21] is a general-purpose programming language designed for multithreaded parallel computing. Cilk is a C/C++ extension that supports nested data and task parallelism.

Frameworks Designed for Compute-Intensive Applications. *Apache Hadoop* [22] is a framework that enables the distributed processing of large data sets across clusters of computers using a simple programming model. *Apache Spark* [23] is a fast and general-purpose cluster computing system.

Scripting Languages for Workflow. *Swift* [24] is a featured data-flow oriented coarse grained scripting language, which is designed for scientists, engineers, and statisticians that need to execute domain-specific application programs many times on large collections of file-based data. *Swift/T* [25] is the high-performance computing version of Swift languages, in which the Swift programs are translated in MPI based programs to be executed on HPC systems. Swift and Swift/T provide set-up on cloud IaaS[1]. *OpenMole* [26] offers tools to run, explore, diagnose and optimize numerical models, taking advantage of distributed computing environments.

[1] http://swift-lang.org/tutorials/cloud/tutorial.html.

Listing 1.1: PI Montecarlo Estimation on Amazon AWS

```
var aws = [type:"aws",access_key:"amazon_aws_access_key",
     secret_key:"amazon_aws_secret_key", region:"us-east-2"]
var ch =  [type="channel"] on aws
func hit(){
   var r =  [type="random"]
   var x = r.nextDouble()
   var y = r.nextDouble()
   var msg=0
   if( (x*x)+(y*y) < 1.0 ){ msg=1 }
   ch!msg
}
func estimation(){
   var sum = 0
   var crt = 0
   for i in [0:10000] {
    sum += ch? as Integer
    crt += 1
   }
   println "PI approximation is "+ (sum*4.0)/crt
}
fly hit in [0:10000] on aws thenall estimation
```

4 FLY Design

The goal of FLY is to provide a portable, scalable and easy-to-use programming environment for scientific computing. FLY perceives a cloud computing infrastructure as a parallel computing architecture on which it is possible to execute some parts of its execution flow in parallel. FLY enables the domain developers (i.e., domain experts with limited knowledge about complex parallel and distributed systems) to design their applications exploiting data and task parallelism on any FaaS architecture. This is achieved by a rich language that provides domain-specific constructs, that enable the developers to easily interact, using an environment abstraction, with different FaaS back-ends.

FLY provides implicit support for parallel and distributed computing paradigms and memory locality, enabling the users to manage and elaborate data on a cloud environment without the effort of knowing all the details behind cloud providers API. A FLY program is executable either on a SMP or a cloud infrastructure (supporting FaaS) without a deep knowledge of the underlying computing resources.

FLY is compiled in native code (Java code) and it is able to automatically exploit the computing resources available that better fit its computation requirements. The main innovative aspect of FLY is represented by the concept of FLY *function*. A FLY function can be seen as an independent block of code, that can be executed concurrently. FLY functions can be executed in sequential mode, in parallel on SMP or on a FaaS back-end. The language provides programming constructs for functions definition, execution, synchronization and communication. Communication among different environments/back-ends is obtained through some virtual communication path named *channels*. Along these lines FLY has been designed as an enhanced scripting language and is composed by a sequence of standard instructions integrated with a number of FLY functions invocation, which interact via channels.

Figure 1 depicts the FLY compilation workflow. On the left side, the FLY program is given in input to the compiler (written using XText). The intermediate AST representation is translated in a Java native program. Each FLY function is translated into different executable codes (one for each back-end). Therefore FLY provides compiled functions code that can be executed on each cloud infrastructure back-end (see Fig. 1).

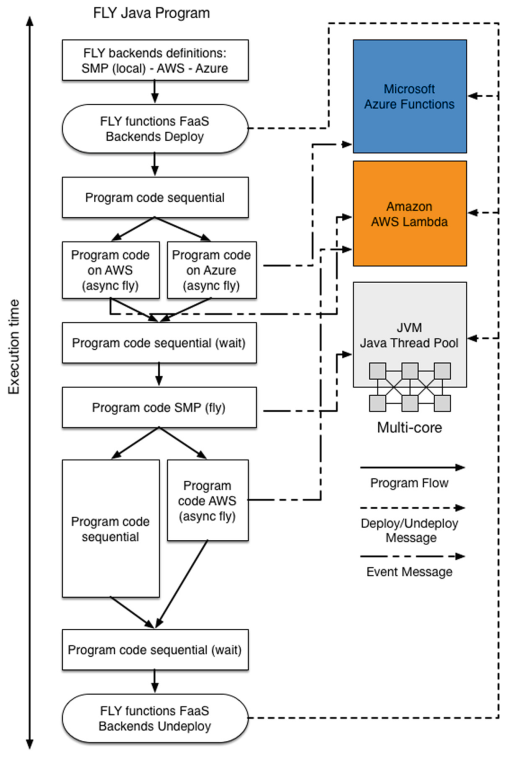

FLY Java Program
Execution time
FLY backends definitions: SMP (local) - AWS - Azure
FLY functions FaaS Backends Deploy
Program code sequential
Program code on AWS (async fly)
Program code on Azure (async fly)
Program code sequential (wait)
Program code SMP (fly)
Program code sequential
Program code AWS (async fly)
Program code sequential (wait)
FLY functions FaaS Backends Undeploy
Microsoft Azure Functions
Amazon AWS Lambda
JVM Java Thread Pool
Multi-core
Program Flow
Deploy/Undeploy Message
Event Message

Fig. 2. FLY execution workflow.

Figure 2 shows a general execution flow of a FLY program along the execution time. First of all, the program initializes all the back-ends required by the FLY code, and *deploys* the generated code on the corresponding back-end. We notice that the FLY functions are already compiled when the main FLY program is executed, thereby avoiding run-time compilation overheads. After these initialization steps, the main program is executed following the FLY code instructions. Each time the **fly** keyword is used, the program generates *events* on the corresponding SMP and/or FaaS back-end, in order to execute the FLY functions. FLY supports synchronous and asynchronous execution models.

Before presenting the FLY language design, Listing 1.1 shows a simple example of a FLY program, which computes a PI estimation through the Montecarlo Method on an Amazon AWS Lambda back-end. Briefly, the PI Monte Carlo estimation algorithm generates a set of random points on a two dimensional Cartesian systems, and counts the number of points that are inside the positive quadrant of a circle of diameter 1.0 centered in the origin. Then, it computes the estimation of PI as $\frac{S*4.0}{N}$, where S is the number of points inside the positive quadrant of the circle and N is the total number of generated points. First of all, FLY *PI* code defines, at line 1, a new Amazon AWS FaaS back-end. Line 2 declares a new channel on the environment `aws` that enables the main program to communicate with the FLY function `hit`, defined at line 3. The `hit` function generates a random point and evaluates whether it belongs to the circle. This information is sent on the channel `ch`. Another function `estimation` reads the outputs of the function `hit` and writes on the standard output the estimation of PI. Line 20 launches 10000 `hit` functions synchronously on the `aws` back-end. When all functions terminate, the function `estimation` is performed on the SMP back-end. It is worth to notice that FLY functions cannot use variables declared outside the function scope, excepts for variables of type channel (see Sect. 5).

5 FLY Language Definition

The FLY syntax and concepts are inspired by different languages such as Java, JavaScript, Python, and R. This ensures familiarity with most powerful and famous general purposes/data science languages. FLY is statically, strongly typed and uses type inference to determine the initial type of all your variables (using the keyword `var`) and constants (using the keyword `const`). Moreover, FLY provides several domain specific constructs for parallel/distributed task/data based parallelism and supports inter-process (and inter-FLY-functions) communications using channels according to communicating sequential processes (CSP) definition [16].

5.1 Data Models and Types

FLY provides two sets of types named *basic* and *domain* types. Basic types, inherited by Java, comprises *boolean*, *integer*, *real* (double point precision floats) and *string*. Moreover, FLY supports one/bi/three-dimensional array definition for basic types. In addition to basic types, FLY provides several domain types that enable the users to interact and communicate with the computing back-ends.

Object Domain Type. The main domain type is the **`object`** type. A FLY object is a heterogeneous collection of basic and/or domain types elements. Essentially a FLY object is a mixture between an array and map data structure, which stores data in key/value pair. The value of an element can be accessed in two different ways: by position (like arrays) or by key (like maps). When a new value is assigned to a given key/position a new element is created, otherwise the new value replaces the previous one. Moreover, all FLY domain specific type are an instance of the object type, which means are build in similar fashion, specifying the object type using the parameter keyword *type = "object_type"*.

Environment Domain Type. The **`Environment`** type represents an abstraction of a execution environment. It provides the ability to interact with a cloud provider or a SMP system. Different environments are treated in the same way by FLY, leaving the details relating to the specific use of each execution environment to the FLY compiler.

Environments are declared as an object using several parameters that characterize a back-end. In this preliminary version of the FLY compiler, the SMP (using the type smp) and AWS back-end (using the type aws) are supported (see Sect. 6).

```
var name = [type="(smp,aws,...)", nthreads=Integer, accesskey=String,
    secretkey=String, limit=Float]
```

The first parameter specifies the desired back-end. The simplest back-end is *smp*, and enables the user to exploit the local SMP architecture. The second parameter (*threads*) indicates the maximum number of concurrent tasks allowed on the back-end. The remaining parameters are used to manage the authentication on the back-end. Eventually, the parameter *limit* enables the user to set an usage cost limit according to the used back-end.

File Domain Type. **`File`** object is the abstraction of file in FLY. The language supports four file formats: *csv*, *json*, *img*, and *txt*, defined by the parameter *type*. A new file object is defined using also additional parameters: *path* (the file system path) or a reference to the file, and by the separator *sep*, that is an optional parameter defined for CSV files.

The language provides two methods to access files, which depend on where the file is stored: *local* or *remote*.

```
var name = [type="(csv,json,img,txt)", path=String, sep=String] on env (optional)
```

FLY has a specific focus on *csv* files managing them as a Dataframe (similar to R language dataframes). The memory is seen as a matrix structure, allowing the user to access to rows and columns, while it provides dedicated operations for querying, filtering, random access, etc. Dataframe operations are described in details in the language documentation.

Communication Statement. **`Channel`** type is a domain type that enables the synchronization and communication between FLY functions and/or the main program, defined by the *type* = *"channel"*. Channels follow the Communicating Sequential Processes (CSP) definition [16]. A new channel is defined on an environment, and can be used for the communication between functions executing on the same back-end or from the main program to a back-end and viceversa. Channels are blocking message queues, that is, when the main program or a function tries to receive a message from a channel, the execution is blocked until a new message arrives on the channel. Messages are sent on a channel using the character '`!`' (e.g., the instruction `ch!VAL` sends a message VAL on the channel ch), while the character '`?`' is used to receive messages, (e.g., the instruction `x=ch?` reads a message from the channel ch and assigns the obtained value to the variable x). Channels use network infrastructures to communicate with the cloud environment and for this reason a serialization mechanism is required for sending/receiving messages. FLY defines the serialization for objects, files, images and basic types. It is not allowed to send messages containing environments, channels, and random objects.

5.2 Control Structures

FLY conditional and iterative controls structures are standard and follows the same statements of languages like Java. Two kinds of *for* loops can be used in FLY, the former uses a range definition, and enables the program to loop in a range of integer values, defined using square parenthesis ([x:y]). The latter, enables the program to iterate over a FLY object or a file.

5.3 Execution Control Structures

Functions. FLY functions are quite different from other scripting languages and follow a functional programming inspired definition. A FLY function represents a task or independent job of the main program and it is defined as a code block that can be executed concurrently. FLY functions are declared using the keyword **`func`**. Each FLY function can have a set of input parameters and may return a value using the word **`return`**. FLY functions have a private scoping, that is only function parameters and local variable are visible in the body of the function. The input parameters are passed by copy, and they are considered as immutable. However, functions can avoid this limitations using channels or constants. A channel declared in the main program or in a function running on the same environment can be directly used by a function, the same behavior is also defined for the constants.

Notice that, the FLY language does not ensure that operations are admitted: if a function is executing on a back-end B, the function can use only channels and objects available on the back-end B. FLY functions can be executed, like for standard languages, using their ID and parameters (in this case functions are executed sequentially). In order to execute functions concurrently, FLY provides the **`fly`** statement that will be described in the following. The **`fly`** statement is not admitted in the body of a function (i.e., recursion is not allowed).

Parallel/Distributed Statement. The definition of FLY functions is the consequence of the explicit parallel execution model of FLY. The language provides the keyword **`fly`** that enables the user to execute concurrently a set of functions (the number of concurrent functions will depend on the back-end used and the user needs). The **`fly`** statement is similar to the **`for`** statement but the **`fly`** statement allows to specify the back-end environment (using the keyword **`on`**) and, possibly, callback functions.

```
fly ID in [x:y]|Object|File on Env  then ID thenall ID
```

The **`fly`** statement supports two kinds of function callbacks, declared using the keywords **`then`** and **`thenall`**. The *then* callback is executed after each FLY function execution, instead the *thenall* callback is executed after all FLY function executions. *Then* and *thenall* functions have to take only one input parameter that, for *then* corresponds to the return value of a function execution, while for *thenall* is a FLY object containing all the return values obtained by all the function executions.

FLY explores synchronous and asynchronous execution models. The previous construct defines the synchronous mode, in which the main program waits all functions termination. It is possible to execute functions asynchronously using the keyword **`async`** before the fly construct.

Asynchronous Execution. The *async* statement returns a special FLY object, named *async-object*, that enables the user to control and interact with the asynchronous execution. The *async* FLY constructor invocation immediately returns the control to the main program and the execution can continue. The user can control the status of the asynchronous functions invoking the method `status()` on the *async-object* and can wait the termination of all functions using the method `wait()`.

Types Casting. FLY uses a dynamic type checking, that is variable types are automatically inferred at run time during the first assignment. Moreover, FLY typing is strong, the type of a variable cannot change during the execution time. FLY provides support for explicit types casting as in Java and C#. Types casting is admitted on basic and domain types, but it is forbidden on environments and channels.

Native Code. FLY is also able to include external libraries (using the keyword **`require`**, which enables to include and install, in the selected environment, an additional library) and supports the execution of native code (using the keyword **`native`**). For instance, the FLY functions running on the *aws* back-end are translated in Javascript, which means that it is possible to include in these functions all JS libraries.

6 Compiler Implementation

We present, in this Section, the preliminary version of our source-to-source compiler for FLY language. An implementation of the language grammar and code generators for the SMP and AWS FaaS back-ends have been developed.

Cloud computing infrastructures expose their FaaS service model through APIs in several languages, as show in Table 1. We deployed our compiler in order to generate a Java program, which is able to support all back-ends. We decided to design our language compiler using *Xtext* [14], which enables the user to create JVM based DSL. The FLY code is translated in a pure Java program that exploits FaaS APIs in order to use FaaS services. Xtext leverages the powerful ANTLR parser which implements an LL parser.

We designed an LL grammar for FLY language that provides the complete language definition, presented in the Sect. 5. Xtext has been also used to develop a code generator that, given the intermediate AST program representation (the output of the first compilation phase), generates a FLY Java program. The code generation phase is the core of our compiler, it generates different codes according to the back-end where the FLY code has to be executed. The code generation phase is designed to be specialized according to the considered back-end:
1) SMP back-end. A Java Thread Pool is used to implement the back-end for the SMP architecture. The FLY main program is executed as Java code on a JVM, which executes also the SMP back-end. In details, all FLY types are mapped on a particular Java type and the FLY functionality are provided exploiting the Java language.
2) FaaS back-ends. The back-ends for Faas architectures have been developed using the Java API of each cloud providers. In order to support different back-ends, our FLY compiler translates each FLY function in JavaScript (JS) using the specific JS cloud provider API to realize FLY operations on channels and remote files. For each back-end and each FLY function the compiler generates a binary package containing: the JS code and the used JS libraries. The generated package is used to deploy the function code on a cloud provider. The alpha release of the FLY compiler as well as the compiler guide is available for download on the GitHub github.com/spagnuolocarmine/FLY-language releases page. The FLY compiler produces: a Java Maven project including all FLY dependencies, the FLY main program (a Java class with the same name of the FLY source code), and the FLY functions code.

7 Conclusion

This paper introduces FLY, a domain specific language for scientific computing on FaaS cloud computing service model. The contributions of this paper are: (i) the design of FLY, a novel domain-specific scripting language for computing-intensive scientific applications; (ii) the language design and specification for SMP and four FaaS cloud computing architectures, and (iii) the FLY source-to-source compiler. Future works and studies are already planned to improve FLY language definition including: library and namespaces definitions, compiler optimizations (according to the FaaS execution model), derived data types (as Java class) and data visualizations. The actual version of the FLY compiler will be extended in order to support the improvements on the language definition as well as other cloud providers. We plan to extend the compiler in order to generate function code in other FaaS languages like Python. Furthermore, FLY will provide specific libraries of algorithms (optimized for FaaS environments), such as machine learning, data mining, and discrete-event simulation. In particular we will focus on graphs algorithms and mining providing support for big networks [1].

References

1. Antelmi, A., Cordasco, G., Spagnuolo, C., Vicidomini, L.: On evaluating graph partitioning algorithms for distributed agent based models on networks. In: Hunold, S., et al. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 367–378. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_30
2. Shawish, A., Salama, M.: Cloud computing: paradigms and technologies. In: Xhafa, F., Bessis, N. (eds.) Inter-Cooperative Collective Intelligence: Techniques and Applications. Studies in Computational Intelligence, vol. 495, pp. 39–67. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-35016-0_2
3. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media Inc., Sebastopol (2016)
4. Cordasco, G., Scarano, V., Spagnuolo, C.: Distributed MASON: a scalable distributed multi-agent simulation environment. Simul. Model. Pract. Theory **89**, 15–34 (2018)
5. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**, 107–113 (2008)
6. Hwang, K., Dongarra, J., Fox, G.C.: Distributed and Cloud Computing from Parallel Processing to the Internet of Things (2011)
7. Baldini, I., et al.: Serverless computing: current trends and open problems. In: Chaudhary, S., Somani, G., Buyya, R. (eds.) Research Advances in Cloud Computing, pp. 1–20. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-5026-8_1
8. McGrath, G., Brenner, P.R.: Serverless computing: design, implementation, and performance. In: ICDCSW 2017, pp. 405–410 (2017)
9. Stigler, M.: Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud. Apress (2017)
10. Groff, J., Weinberg, P.: SQL The Complete Reference (2010)
11. Graham, I.S.: The HTML SourceBook. Wiley, New York (1995)
12. Cosenza, B., et al.: OpenABL: a domain-specific language for parallel and distributed agent-based simulations. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) Euro-Par 2018. LNCS, vol. 11014, pp. 505–518. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96983-1_36
13. R Development Core Team, R: A Language and Environment for Statistical Computing (2008). www.R-project.org
14. Eclipse Project, Xtext, Language Engineering For Everyone! (2018). www.eclipse.org/Xtext
15. Thoman, P., et al.: A taxonomy of task-based parallel programming technologies for high-performance computing. J. Supercomput. **74**, 1422–1434 (2018)
16. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall, London (1997)
17. Rossum, G.: Python reference manual. Technical report (1995)
18. Bezanzon, J., Karpinski, S., Shah, V., Edelman, A.: Julia: a fast dynamic language for technical computing. In: Lang.NEXT (2012)
19. Ritchie, D.M.: The limbo programming language. Technical report (2018)
20. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. **21**, 291–312 (2007)

21. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. **30**, 207–216 (1995)
22. White, T.: Hadoop: The Definitive Guide. O'Reilly Media Inc., Sebastopol (2009)
23. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**, 56–65 (2016)
24. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: a language for distributed parallel scripting. Parallel Comput. **37**, 633–652 (2011)
25. Wozniak, J., Armstrong, T., Wilde, M., Katz, D.S., Lusk, E., Foster, I.: Swift/T: large-scale application composition via distributed-memory dataflow processing. In: Proceeding of IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (2013)
26. Reuillon, R., Leclaire, M., Rey-Coyrehourcq, S.: Openmole, a workflow engine specifically tailored for the distributed exploration of simulation models. Future Gener. Comput. Syst. **28**, 1981–1990 (2013)