2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)

Visual-textual framework for serverless computation: a Luna Language approach

Piotr Moczurad*†, Maciej Malawski*
*AGH University of Science and Technology,
Department of Computer Science,
Krakow, Poland,
Email: malawski@agh.edu.pl
†Luna Language, www.luna-language.org,
Krakow, Poland,
Email: piotr.moczurad@luna-lang.org

***Abstract*—As serverless technologies are emerging as a breakthrough in the cloud computing industry, the lack of proper tooling is becoming apparent. The model of computation that the serverless is imposing is as flexible as it is hard to manage and grasp. We present a novel approach towards serverless computing that tightly integrates it with the visual-textual, functional programming language: Luna. This way we are hoping to achieve the clarity and cognitive ease of visual solutions while retaining the flexibility and expressive power of textual programming languages. We created a proof of concept of the Luna Serverless Framework in which we extend the Luna standard library and we leverage the language features to create an intuitive API for serverless function calls using AWS Lambda and to call external functions implemented in JavaScript.**

I. INTRODUCTION

Serverless computing has several traits that make it the topic of interest in the cloud computing community. Some of the most obvious are:

1) the ability to cut down on the cost of running the infrastructure due to a highly elastic, on-demand model of computation and pricing,
2) abstracting away most of the infrastructural concerns to let the developers focus on the business logic and spare them the work required to maintain a separate virtual machine in the cloud.

However, the flexibility comes at a cost: once one starts splitting out the logic of the whole system into separate functions (as opposed to maintaining a few, clearly separated services), the responsibilities and dependencies between different parts of the system are becoming obfuscated and the subdivision of the system is less clear. The way the serverless functions are defined, the programmer is unable to get the feeling of the flow of data through the system. Consequently, while the costs are reduced and it is easier to construct the system and bring it online, it is also easier to make a mistake.

In this paper, we argue that one promising way of managing the complexity induced by scattering the computation into small remote functions is by letting the programmers visualize the system and, importantly, the flow of data between the functions. We describe a way of integrating the serverless computations tightly into the Luna programming language, leveraging some of its features: dual, visual and textual representation, expressive syntax, functional semantics and a modern static type system. In Section II we discuss the serverless technology, with its benefits and pitfalls. Then we move on to the Luna programming language in Section III, describing its features and their relation to the serverless computation. Section IV is a presentation of the Luna Serverless framework: a presentation of features and a technical overview. Section V presents the evaluation of the solution and Section VI describes the planned and possible future work.

II. SERVERLESS COMPUTATION

A. The trend

Observing the trends in the cloud computing space, one quickly notices that the models of computation evolve as more and more of the infrastructure is abstracted away and hidden from the eyes of developers. Back in a day one would maintain own hardware, on which the servers ran. This had a major disadvantage in terms of scalability: the limited processing power could not handle any spikes of the load. Consequently, the companies started moving away from maintaining their own datacenters and towards renting out virtual machines in the cloud in the Infrastructure as a Service model. This approach is more scalable and allows the company to spend less time managing the infrastructure. However, as the services started getting split on the rising wave of popularity of the Service Oriented Architecture and the microservices trend, it became apparent that the orchestration and configuration of the services is cumbersome and can be abstracted away. The serverless architecture is an almost perfect abstraction, in which there is virtually no infrastructure to manage and the developer's only concern is to choose which functions to run and how to design the business logic. Recently, the serverless paradigm has drawn attention of the scientific community with some research proving its usefulness not only in business applications but in large-scale computing [12, 13].

B. The shortcomings

As promising as the serverless model may appear, it has major disadvantages that prevent its broader adoption. First

978-1-7281-0359-4/18/$31.00 ©2018 IEEE
DOI 10.1109/UCC-Companion.2018.00052

of all, splitting a computer system into a number of remote function invocations entails significant complexity: the system is no longer comprised of a few blocks with easily separable responsibilities. Instead, it becomes a tangled spaghetti of interleaving calls, making it hard to comprehend and debug. We observe that the paradigm most often adopted by the serverless applications is similar to the event-driven model very popular amongst web application developers. Just as the event-driven architecture can lead to a callback hell, the serverless applications can get overly obfuscated. This is especially problematic given the lack of appropriate tooling that would facilitate visualizing and testing the flow of data through the system [1, 3]. Additionally, composition of serverless functions is non-trivial and gives rise to problems such as the tradeoffs described in [4]. Secondly, the abstraction that the serverless provides is not complete: the configuration is still more cumbersome than it has to be, involving configuration files that vary from platform to platform. This makes programming harder and locks the application to a single vendor [2].

C. The trend continued

A natural development of the trend we have briefly described would be to integrate the serverless computations tightly into a programming language, thus hiding all of the configuration from the programmer. Once the remote functions are almost indistinguishable from the functions one calls locally, the development of serverless applications will be as productive as creating single-machine programs. It would make sense for the language to be functional, as the serverless computations inherently impose a functional mindset onto the developer, forcing them to manage the application state in a similar fashion as one does when writing functional programs. The ease of (possibly formal) reasoning about the functional programs directly addresses some of the concerns above. Using the web application analogy, it is analogous to the introduction of Functional Reactive Programming and languages such as ReactML [10] and PureScript [5]. As described in later sections, when combined with visual programming, most of the aforementioned problems are addressed.

D. Related work

There already exist solutions trying to manage the complexity of serverless applications, majority of them attempting to visualize the workflows. They are, however, different in several important aspects, the most profound difference being: none of the solutions tries to integrate the serverless workflow into a general-purpose programming language. We will briefly outline some of the best solutions available.

Amazon Web Services Step Functions [9] is a tool for creating state machines based on the serverless architecture and microservices. The tool features a visual representation of the state machine but the interaction with the system is still predominantly text-based, with quite a lot of configuration. The visual representation is only passive: it cannot be used to create the state machines, only to inspect them. Step Functions allow for easy monitoring of the execution of the functions involved.

IBM Composer [8] is a framework facilitating composition of IBM Cloud Functions, featuring a non-interactive visual representation of the resulting graph. It offers function combinators that allow for composing more elaborate workflows from simple functions (e.g. conditional execution). What makes it similar to functional languages is the stress put on the notion of composition, which is one of the key aspects of functional software that make it easy to extend and debug. One of the drawbacks of the framework is its use of JavaScript, which exposes the users to many of its pitfalls and makes refactoring a painful process.

Malawski et al. [14] describe a framework for creating scientific workflows based on the serverless technology which, unlike the previous solutions, is not tied to a specific vendor. The resulting workflows are conceptually quite similar to the graphs in visual languages: however, it is not possible to interact with them and they are not a first-class citizen of the framework. This is understandable given the scientific and supercomputing provenance of the framework: it was created by experts, for experts, with an average programmer not being a part of the key demographic. The very important result of their work is proving that serverless bears incredible potential for computations until recently requiring a supercomputer.

III. The Luna Language

In this section we introduce Luna [18], a visual-textual, purely functional programming language and discuss how it can fit into the serverless ecosystem. The purpose of Luna is to introduce a novel programming paradigm, blending the dataflow style with purely functional traits. One field in which Luna is already used is Business Intelligence (mainly data discovery and visualization). We present Luna's features in context of its adoption in the distributed computing industry, and more specifically: serverless computations.

A. Visual-textual representation

The Luna programming environment is built around the concept of two representations of the program: one is the orthodox, textual representation that every programmer is familiar with. The other one is the visual representation in form of a dataflow graph. This has been successfully adopted by the visual effects industry for a few decades now and is the key to making the programs accessible to a wider audience and easily comprehensible.

One of the key features of this dual representations is the real-time way in which Luna maintains the isomorphism between the text and the graph. Everything that is added to the graph is immediately reflected by the program's text. Conversely, any word typed in the text representation has its reflection in the graph. Consider the following code:

```
x = 2
y = 2
sum1 = x + y
```

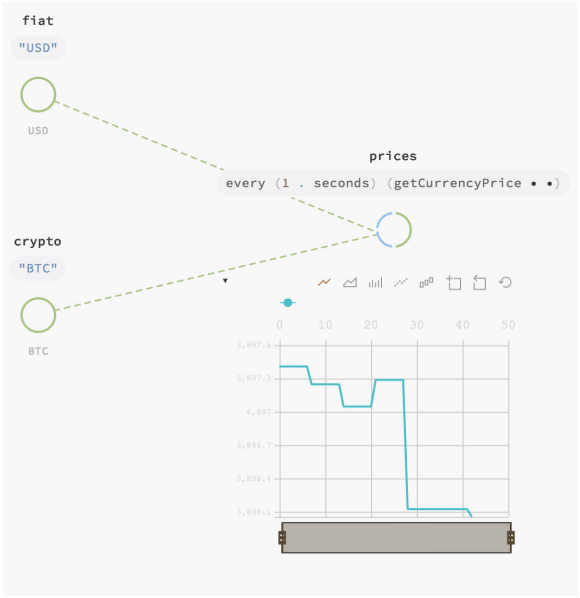

Figure 1. Real-time visualisation of cryptocurrency prices.

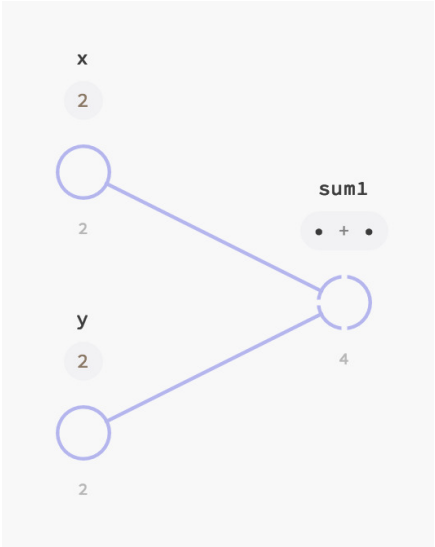

Figure 2. Simple arithmetic operations in Luna.

The resulting graph is depicted in Figure 2. This allows the programmer not to commit to either representation and freely choose the more convenient representation for the problem at hand. It is worth noting that usually it is beneficial to prototype and inspect the high-level overview of the system (learning the flow of data) and to tweak the implementation details using the text representation, which allows for an in-depth tuning of the program.

B. Programming paradigm

Luna encourages the use of purely functional programming in both representations. In a way, a computation graph is an inherently functional construct, with data flowing from node to node, undergoing subsequent transformations that are referentially transparent: a node never changes its inputs and always returns a result. As far as "impure" computational effects are concerned, Luna tries to adopt a functional approach towards their handling, alas without burdening the programmer with concepts like monad transformers or algebraic effects: the compiler figures out the enclosing monads by itself, leaving the code clean and the programmer unburdened.

The textual representation syntax is conceptually quite similar to purely functional languages like Haskell but with some features added (e.g. objects) and much of Haskell's complexity stripped down. One of the factors that strongly influence the textual representation is the need to maintain its isomorphism with the graph. An interesting example of that is the variant of lazy semantics adopted by Luna: each line is evaluated by default, but nested structures are only "unwrapped" one layer deep (this is similar to evaluating an expression to the Weak Head Normal Form) [7]. This gives rise to the following behavior:

```
# fully evaluated, x contains the text user inputs:
x = Stdin.read

# the line is evaluated, but anything under
# the constructor remains a thunk
# (Just is a simple type constructor with
# a sole purpose of being a placeholder):
y = Just Stdin.read

# a call that unpacks the placeholder
# causes the inner value to be evaluated:
z = y.get

# subsequent upackings will evaluate
# the action each time:
w = y.get
```

This approach allows for easy control over when an expression will actually get evaluated. In languages with full lazy evaluation it can be non-trivial to say when an expression will be evaluated (if it will be, at all). Consequently, exception handling becomes an involving task and performance optimizations require deep knowledge of the semantics. Luna is an attempt to retain the benefits of lazy semantics without introducing caveats.

C. Luna in the serverless world

We notice that the way in which Luna tries to solve problems encountered in a general programming setting can be extended to address the problems of the serverless computation model and its tooling. Once integrated with the language, the visual representation provides the visualization facilities for the serverless architecture for free, leveraging Luna's built-in capabilities. Moreover, integration with a functional language encourages adopting a more functional style when developing applications. This facilitates creation of software that is correct and easy to reason about.

IV. Luna Serverless Framework

A. Initial remarks

The framework for serverless computation created for the Luna programming language (the Luna Serverless Framework, abbreviated LSF) is based around the concept that the remote calls should be as similar to regular function calls as possible. In Luna, there exist two possible ways of doing so:

1) Extend the compiler to acknowledge the existence of functions executed in non-local contexts, such as serverless.
2) Extend the standard library and leverage the language features to create an intuitive API for serverless function calls.

For the first version of the Luna Serverless Framework we decided to follow the second approach. It is better suited for a proof-of-concept and does not require deep changes to the compiler. However, we are making our way towards full integration into the language, as described in Section VI. Also, the first version integrates only with the Amazon Web Services Lambda. The choice was motivated by the platform being a de-facto standard for serverless applications, as described by [11]. Note however, that one of the existing drawbacks of serverless solutions we are trying to address is the vendor lock-in. As such, the coming versions of LSF will be adding other integrations, like Google Cloud Functions and Microsoft's Azure Functions. That said, all of the ideas we present are vendor-agnostic and apply to any flavor of the serverless model.

B. API walkthrough

The framework can be divided into four parts:

1) Environment initialization
2) Function creation
3) Function invocation
4) Miscellaneous utilities

We will present each part by presenting a very simple, example workflow using LSF. Each step will present the code and the corresponding graph. Note that the key feature of LSF and Luna in general is that there is no need to commit to either representation and they can be used interchangeably. We provide both partly to showcase the dual representation and partly to let the reader choose the form the deem more comprehensible.

All of the LSF code is located inside the `Std.AWS` module. It needs to be imported, which is done using the `import Std.AWS` command. When writing Luna programs, it is also recommended to import the `Std.Base` module which exposes common types and operations.

1) Environment initialization: All of the cloud services require some groundwork before they are usable: authentication, establishing of user roles, permissions, etc. In LSF this is done by creating a configuration object.

```
aws = AWS.init
```

Figure 3. Initialization of the AWS environment: the `AWS` class contains all of the methods for interacting with Lambda and therefore is the "starting point" for all computations.

The AWS part of the Luna Serverless Framework uses the Haskell library Amazonka [6] to issue AWS API calls and borrows its authentication flow: if it exists, the credentials will be read from the standard configuration file (`~/.aws/credentials`). If not, the environment variables `AWS_SECRET_KEY` and `AWS_ACCESS_KEY` will be used. If they are not set, an error will be thrown. If the initialization was successful, an `AWS` object is created, as seen in figure 3. It will allow for creation of the functions and issuing other commands. If we want to proceed with function creation, we need to do one more thing: set the IAM role. We decided to leave the control over the roles to the user so that the framework does not do too much behind the user's back. That, however, might change in the coming releases.

```
aws2 = aws.setRole "YOUR IAM ROLE"
# or, using a more concise syntax:
aws = AWS.init . setRole "YOUR IAM ROLE"
```

2) Creating functions: When declaring a function, two things are necessarry: the function's name and its code. In the current version of the framework, the functions need to be written in vanilla JavaScript (even though eventually one will be able to write the serverless code in Luna). The reason for choosing JavaScript is its widespread adoption within different serverless implementations.

There are two ways of supplying the code for the function: directly, as a string/text (using the `LambdaFunctionCode.fromText` method), or from a file (using `LambdaFunctionCode.fromFile`). Both of these return a `LambdaFunctionCode` instance that can be used during function definition. Once we have the code, we need to supply it to the `createFunction` method of the `aws` object we created in the previous step. The `AWS` class maintains a cache of functions to make sure that they are not created many times when the program is re-evaluated. All in all, it looks like the following:

```
text = "exports.handler =
    async (event) => {
        return 'Hello ' + event.name + '!';
    };"
code = LambdaFunctionCode.fromText "testFunc" text
(aws2, testFunc) = aws.createFunction code
```

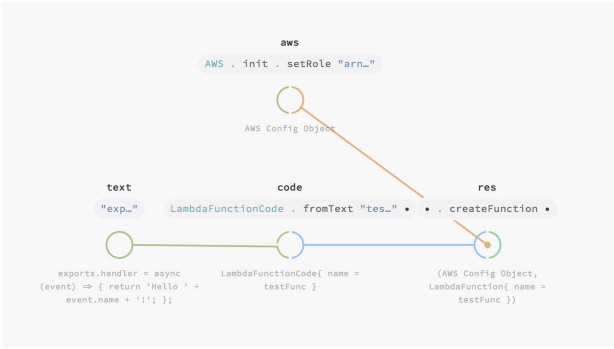

Figure 4. Definition of a Serverless function.

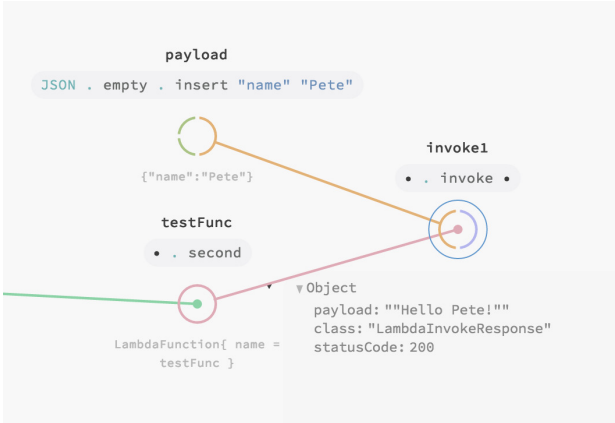

Figure 5. Invocation of a Serverless function.

The listing above can be written all in one line, additional bindings were added for the reader's convenience. The `createFunction` calls returns a `LambdaFunction` instance that encapsulates the remotely-defined function and a new configuration object that is aware of the function we just added. Note that we are returning a new instance of this object (instead of modifying it in place) to adhere to the functional and dataflow programming styles.

3) Function invocation: The `LambdaFunction` class has a `invoke` method that is used to call the functions. The argument is a JSON payload. In Luna, one would do it as follows:

```
payload = JSON.empty . insert "name" "Peter"
res = testFunc.invoke payload
```

The `res` variable is of type `LambdaInvokeResponse`, exporting methods such as `text`, `json` or `statusCode` for inspecting its contents. Note that by default, in the visual representation the result of the response will be immediately visible under the node, without the need to explicitly deconstruct the response.

4) Miscellaneous utilities: The `AWS` class, which was used to create a function, also provides some utilities for dealing with the functions, such as listing every available function or deleting functions.

V. Evaluation

The solution was meant to serve as a proof-of-concept and to pave the road for further integration of the serverless technologies into the Luna programming language. However, we were able to create a framework that is both usable and provides a clean and extensible API.

The framework described in this paper is open source and its source can be found on GitHub in the Luna repository [16] and in the Luna Studio repository [17]. We tested the framework with sample AWS Lambda functions and it is able to successfully connect to AWS, authenticate, configure the environment, define the functions and invoke them with different payloads. The performance of the solution is comparable to Python implementations of remote invocations of functions, but we have not yet performed detailed experiments to measure it systematically. We assume that the upper bounds on the execution time are dictated predominantly by AWS Lambda, its response times and the general latency involved in communication over the HTTP protocol. The Luna language itself is performing similarly to Python and thus is by no means a bottleneck in such context.

In terms of developer productivity, the improvement can be potentially considerable: first of all, the type system allows for early detection of developer errors and the visual workflow is intuitive and designed for ease of use. However, further studies on the productivity are needed to fully assess the benefits of our approach.

VI. Future work

As stated before, the Luna Serverless Framework is in its proof-of-concept phase and hence under constant development. There is a clear roadmap where we would like it to go, moving it towards tighter integration with the Luna programming language and environment.

First of all, the remote functions need to become a first class citizen of the language. It is our belief that a function on AWS Lambda should differ from a locally-defined function only in terms of its type. One proposed syntax for defining remote functions is the following:

```
remote def testFunc payload:
    # function code
```

Second, the functions need to be written in Luna itself to fully leverage its functional and visual aspects. This is a major project in itself and will require creating a JavaScript backend for Luna.

Third, we plan more detailed performance and productivity studies of Luna Serverless Framework. It will allow us to fully evaluate the advantages and limitations of our approach.

Lastly, we would like to create a fairly rigorous theoretical framework for reasoning about serverless applications written using the Luna Serverless Framework and leverage the benefits

of the functional paradigm and advanced type system: it will most probably be based on a form of a process calculus like the π-calculus [15] or one of its extensions.

We strongly believe that with the improvements listed above, the serverless model could see a more widespread adoption and gain many new target audiences.

ACKNOWLEDGMENTS

The authors would like to thank the New Byte Order company for lending their infrastructure and cloud resources to conduct numerous experiments using the Luna Serverless Framework. This work was also supported by AGH Dean's Grant.

REFERENCES

[1] Rohit Akiwatkar. *The Drawbacks of Serverless Architecture*. URL: dzone.com/articles/the-drawbacks-of-serverless-architecture. (accessed: 30.08.2018).

[2] Matt Asay. *The 2 biggest problems with serverless computing*. URL: www.techrepublic.com/article/the-2-biggest-problems-with-serverless-computing. (accessed: 30.08.2018).

[3] Ioana Baldini et al. "Serverless Computing: Current Trends and Open Problems". In: (Jan. 2017), pp. 1–20.

[4] Ioana Baldini et al. "The Serverless Trilemma: Function Composition for Serverless Computing". In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. Vancouver, BC, Canada: ACM, 2017, pp. 89–103. ISBN: 978-1-4503-5530-8. DOI: 10.1145/3133850.3133855. URL: http://doi.acm.org/10.1145/3133850.3133855.

[5] Phil Freeman and open source contributors. *PureScript*. URL: purescript.org.

[6] Brendan Hay and open source contributors. *The Amazonka Haskell library*. URL: github.com/brendanhay/amazonka.

[7] Paul Hudak et al. "A History of Haskell: Being Lazy with Class". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007, pp. 12-1–12-55. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856. URL: doi.acm.org/10.1145/1238844.1238856.

[8] IBM and open source contributors. *IBM Functions Composer*. URL: github.com/ibm-functions/composer.

[9] Amazon Web Services Inc. *AWS Step Functions*. URL: aws.amazon.com/step-functions.

[10] Facebook Inc. *Reason ML*. URL: reasonml.github.io.

[11] T. Lynn et al. "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms". In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Dec. 2017, pp. 162–169. DOI: 10.1109/CloudCom.2017.15.

[12] Maciej Malawski. "Towards Serverless Execution of Scientific Workflows - HyperFlow Case Study". In: *WORKS@SC*. 2016.

[13] Maciej Malawski et al. "Benchmarking Heterogeneous Cloud Functions". In: *Euro-Par Workshops*. 2017.

[14] Maciej Malawski et al. "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions". In: *Future Generation Computer Systems* (2017). ISSN: 0167-739X. DOI: doi.org/10.1016/j.future.2017.10.029. URL: www.sciencedirect.com/science/article/pii/S0167739X1730047X.

[15] Robin Milner, Joachim Parrow, and David Walker. "A calculus of mobile processes, I". In: *Information and Computation* 100.1 (1992), pp. 1–40. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90008-4. URL: http://www.sciencedirect.com/science/article/pii/0890540192900084.

[16] Piotr Moczurad and the Luna Team. *The aws-serverless branch of the Luna Language repository*. URL: github.com/luna/luna/tree/aws-serverless.

[17] Piotr Moczurad and the Luna Team. *The aws-serverless branch of the Luna Studio repository*. URL: github.com/luna/luna-studio/tree/aws-serverless.

[18] The Luna Team. *The Luna Programming Language*. URL: www.luna-lang.org.