Toward a Multi-Language and Multi-Environment Framework for Live Programming

Hidehiko Masuhara Shusuke Takahashi Yusuke Izawa Youyou Cong
masuhara@acm.org,{takahashi,izawa}@prg.is.titech.ac.jp,cong@c.titech.ac.jp
Tokyo Institute of Technology

Abstract

While applications of live programming are expanding to more practical and professional domains, most live programming environments (LPEs) are still developed for a single target language with an original code editor. We propose an implementation framework for developing LPEs so that we can minimize efforts on implementing an LPE for a different target language and an existing code editor/IDE. Our idea is to use a meta-JIT language implementation framework (e.g., Graal/Truffle and RPython) and LSP to separate core live programming implementations from language-specific and editor/IDE specific implementations. This paper takes the Kanon live programming environment as a concrete example and discusses how we can design the framework to accommodate the features of Kanon. Although the framework design is still underway, the paper presents a sketch of the framework APIs for separating language-specific functions and clarifies the requirements to LSP.

1 Introduction

Live programming is a programming activity where the programmer incrementally writes code fragments in a program while immediately observing the behavior of the program [14]. A live programming environment (LPE) is a code editor or an integrated development environment (IDE) that immediately re-executes the program when it is changed and presents the code's behavior so that the programmer can recognize the effect of the edit from the presented behavior.

While live programming has initially been practiced in specific kinds of programs, such as programming education [4] and musical performance [1], its applications are expanding to more professional programs, including exploratory data analysis [9, 12], high-performance computing [13]and general-purpose programming with data structures [10].

However, there are still challenges to live programming for overcoming the "Real Programmer" [11] syndrome. In other words, live programming still needs to be expanded to wider ranges of domains in the following respects.

- Most LPEs support a single programming language, sometimes a newly designed language (e.g., [5, 6]). Constructing LPEs for the real programmer's favorite languages requires a considerable amount of effort.
- Most LPEs are developed as their own programming environments. Bringing the LP features to the real programmer's favorite programming environments is not easy especially by considering the number of different programming environments used in practice.

The reasons that make LPEs remain in specific languages and environments are that LPEs require tight integration with the target language and the code editor. LPEs require information on program execution that is only available inside of a language runtime, for example, a value of a variable in the middle of program execution. They also need to observe editing activities so that they can immediately present the program's behavior as soon as the program is edited. As a result, those environments often come with a customized interpreter or compiler that is integrated with a newly developed code editor.

This paper proposes a polyglot and ploy-environment LPE implementation framework called *Poly²Kanon* based on our previous work Kanon, a live data structure programming environment for JavaScript [10]. The goal of the framework is to easily implement Kanon for many programming languages and for many existing code editors and IDEs. Our idea is to separate the language and environment neutral implementation from the language runtime and the code editor as illustrated in Figure 1,

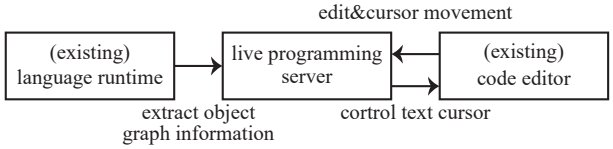

Figure 1. Overview of the *Poly²Kanon* Framework

Presented at Live Programming Workshop 2020 (LIVE'20), co-located with SPLASH 2020 on November 17, 2020.

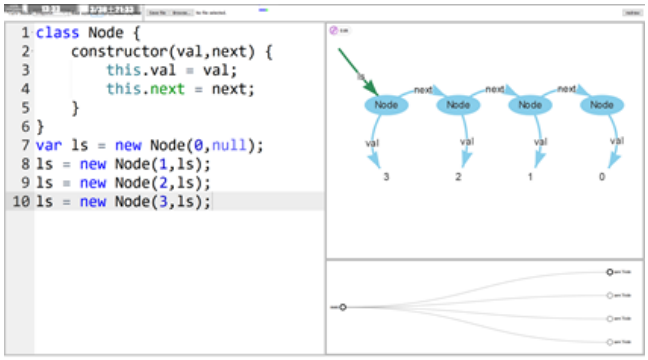

Figure 2. Screenshot of Kanon. The left pane is a code editor where the programmer edits a program. The top-right pane displays the objects created during the execution of the program as a node-link diagram. The diagram will be updated immediately when the programmer modifies the code.

and use a language implementation framework to extract information and use the Language Server Protocol (LSP) [7] in order to interact with many code editor implementations.

We hereafter focus on a specific LPE namely Kanon to make the discussion concrete. However, we believe that the basic idea and many of the specific ideas would also be valid to many types of LPEs.

In the rest of the paper, we first introduce Kanon's features and its implementation, and two core technologies namely language implementation frameworks and LSP as the background of the work. We then discuss the requirements of the language runtime and the editor from the LP engine.

2 Background

This section introduces Kanon's features from the programmer's viewpoint, and its key implementation techniques [10], followed by the introductions to the two existing technologies that we will use in our proposal, namely the language implementation frameworks [2, 15] and LSP [7]. The readers familiar with those technologies may skip to the next section.

2.1 Live Programming Features in Kanon

Kanon is an LPE for JavaScript programs, and lively visualizes object structures as a node-edge diagram (Figure 2). The node-edge diagram (in the top-right pane in the screenshot) represents the state of the objects created during an execution of a user program. As the objects and their relations change during the execution, the displayed diagram is synchronized with the position of the text cursor; i.e., it shows the object diagram of the execution at the text cursor.

When the programmer is about to write a function that manipulates objects, the visualization helps the programmer's Plan-Do-Check process in this way: **(Plan)** the programmer reasons about the relations between objects from the diagram (e.g., "this object is connected to that object by the next field" and plans the next immediate operation (e.g., "change the next of this to a new object"); **(Do)** along with the plan, the programmer writes one or a few lines of code, which trigger Kanon to immediately re-execute the program and to visualize the objects after executing the newly inserted code fragments; and **(Check)** the programmer checks if the code behaved as expected by reading the updated visualization.

Automatic visualization: Kanon automatically collects and visualizes objects without special directions in a program. In contrast, existing LPEs require explicit drawing commands written in a program to give visual feedback [4][1].

Navigation based on the text cursor: Kanon synchronizes the visualized object diagram with the text cursor position. The object diagram also shows the references from local variables that are available at the text cursor position.

Navigation based on the call graph: Kanon also displays a call graph for navigating the visualization (the bottom-right pane in the screenshot). This is useful when a program executes one source code location more than once either by calling a function multiple times or running a looping construct.

Context preservation: Kanon uses calling-contexts for distinguishing execution points matching a source code location. As a result, when the programmer places the text cursor on a source code location that is executed more than once, and he or she moves the cursor to the next line or inserts a code fragment, Kanon will display the object diagram at the new text cursor position *in the same calling context.*

Jump-to-construction: When the programmer clicks on a node or an edge in the object diagram, Kanon moves the text cursor to the source code location that created the clicked object or that assigned the value to the clicked field.

2.2 Implementation Techniques of Kanon

In this section, we overview the current implementation of Kanon.

[1] See our previous paper [10] for further discussion when automatic visualization is appropriate.

Dedicated code editor: The code editor part is extended from the Ace editor [3], an open-source project with basic code editor features like syntax highlighting and automatic indentation. Kanon extended it to detect text cursor movement, and to detect edits of the user program (e.g., insertion and deletion of text).

Source code instrumentation: To automatically collect object graph information from an execution of a program, Kanon instruments checkpointing operations into the user program and let the standard JavaScript engine run the instrumented program.

Checkpointing: Each instrumented checkpoint, given a root set (set of objects referenced by global and local variables accessible from the inserted code location), traverses objects reachable through field references. It records those objects as a copy of a node-edge graph. To traverse objects, Kanon uses the reflection mechanism in JavaScript.

Call stack monitoring: To provide the call graph and implement the mental map preservation, Kanon also instruments code before and after every function call site. The code simulates the call stack, whose information will be recorded in the checkpointing operations.

2.3 Language Implementation Framework

In this section, we introduce *meta-JIT* language implementation frameworks. In general, a language implementation framework is a framework that enables language implementations (compilers, virtual machines, etc.) easier by providing commonly used functions. A meta-JIT language implementation framework, as exemplified by RPython [2] and Graal/Truffle [15], allows the language developer to merely define an interpreter of the target language to obtain the runtime of the language with a just-in-time compiler.

Both RPython and Graal/Truffle are successful in implementing various languages including Python, JavaScript, Smalltalk, Ruby, and R. Those frameworks are also successful in providing quality language runtimes.

2.4 Language Server Protocol (LSP)

The Language Server Protocol (LSP) is a protocol "used between an editor or IDE and a language server that provides language features like auto complete, go to definition, find all references etc." [7] It aims at minimizing efforts on providing language features to different code editors and IDEs. The official website lists 34 tools that support LSP including major IDEs and code editors such as Visual Code Studio, Eclipse, and Sublime Text.

The current specification lists 52 protocols in total. Most of the protocols are commonly found in code editors and IDEs like requesting code completion from the editor to the server. The followings are examples.

hover Request: The editor requests the server to show a piece of hover information at a specific text in the program.

DidChangeTextDocument: The editor notifies changes in the program.

Goto Definition Request: The editor requests the location of an identifier in the program.

3 Language and Environment Specific Implementations

To design a framework for supporting multiple languages and environments, we here analyze an existing implementation of LPE in terms of dependency on the target language and environment.

3.1 Language Specific Implementations

Source code instrumentation It depends on the syntax and semantics of the target language as it needs to insert code into specific syntactic nodes of the parsed program tree. Its implementation is often burdensome as many programming languages have many different syntactic categories. Though there are AST libraries for many languages that provide some means of instrumenting code, there is no single library that uniformly covers multiple languages as far as the authors know.

Checkpointing It depends on the target language in two ways. First, to obtain the root set at each source code location, it needs to understand the variable scoping rule of the language. Second, traversing objects requires a reflection mechanism of the language, which is different between languages.

Logging It is an alternative technique to collect object graph information, though not used in Kanon. The logging technique records every event that modifies an object graph (e.g., creation of an object, and update of a field of an object) during execution of the user program. From the recorded history of events, we can reconstruct an object at any point of execution. This technique is usually realized by modifying an interpreter of the target language, which is definitely language-dependent. It is not easy to realize by using source code instrumentation because operations performed inside of the system library cannot be instrumented.

Call stack monitoring It depends on the target language's functionality. Some languages provide some means of accessing the call stack information (e.g., `thisContext` in Smalltalk) but in a language-specific way. In some other languages that do not provide sufficient information, we need to monitor by code instrumentation, which is again language-dependent.

3.2 Programming Environment Specific Implementations

Although visualization of the object graph can be independent of languages and environments, many other features in Kanon depend on the programming environment.

Navigation of visualization Kanon visualizes the object graph at the text cursor position. It also provides several commands to navigate the calling context. These features require to detect text cursor movement and invocations of commands in the code editor, which depend on the code editor implementation.

Controlling the text cursor position Kanon offers the two features that move the text cursor position to a specific code location based on user actions performed on the visualization. The one is jump-to-construction, and the other is calling context selection on the call graph visualization. Those features need to control the text cursor position from outside of the code editor, which depends on the editor's implementation.

Code change detection Kanon re-executes the user program as soon as it is changed. Moreover, Kanon requires to know the details of the change made on the user program so that it can match the code locations in the older and changed program texts. This requires detailed information from the code editor, which is implementation-dependent.

4 *Poly²Kanon*: a Polyglot and Poly-Environment Framework

We propose *Poly²Kanon*, an implementation framework for live data structure programming. It is *polyglot*, i.e., easily supports multiple languages by exploiting existing language implementation frameworks. It is *poly-environment*, or easily supports multiple code editors or IDEs based on the idea of LSP.

Since the proposal is not yet implemented, we here discuss how the underlying technologies can make the framework polyglot and poly-environment. As an underlying language implementation framework, we assume Graal/Truffle here. However, most of the discussion will also be valid for RPython.

4.1 Sharing Language Implementations Through a Language Implementation Framework

The key idea here is to provide a framework for object graph logging and call stack monitoring so that we can equip those functions into each language runtime with minimal modifications.

We here list some of the framework API functions that are provided to and required by the language developer. The design of the whole API is still underway. The framework provides functions to record events that modify an object graph: `create(object,loc)` and `update(object,field,value,loc)` respectively records construction of an `object` and assignment to a `field` of an `object`, for example. The last parameter `loc` tells the source code location that performed the event.

The framework also requires the language developer to implement functions that depend on the language implementation: `get_class(object)` and `get_field(object)` are typical reflective functions for example. Other required functions are `get_source(...)`, `set_id(object, ID)` and `get_id(object,id)` which returns the source code location of the current execution point, associates and retrieves a unique `ID` of an `object`, respectively.

With those provided and required functions, logging and call stack monitoring will be implemented by modifying a few handlers[2] in the interpreter implementation. For example, the developer will modify a handler for object creation so that it will first obtain the source code location, and then call `create(object,loc)`. The `create` function then generates a unique ID for the object, associates the ID to the object by calling `set_id` function, collects the field values of the object by calling `get_field`, and records the event with the ID, the field values, and the source code location.

4.2 Using LSP for Supporting Multiple Programming Environments

The concept of LSP exactly suits our goal to provide live programming to multiple programming environments. The question is whether there are suitable protocols to implement the Kanon's features. Since each protocol specification only describes types of parameters and a few sentences of the behavior, we would need to investigate each code editor's behavior with respect to each protocol to answer this question (note that editors may not implement all the protocols). We here discuss, based on the protocol specification, our plan to implement Kanon's feature by using the protocol.

Displaying visualization LSP does not define a protocol to display visual information like an object graph alongside the code editor. This is however not a serious problem as the LP server can create its own window to display the graph.

Code change detection LSP defines `DidChange TextDocument` protocol to notify the server when the

[2]By a handler, we here mean the interpreter's implementation for specific kind of operation in the language such as object creation and field assignment.

program text is changed in the editor. The protocol also gives the source code location of the change and the content of the change. It therefore should be sufficient to implement the re-execution feature and to match code locations in the older and the newer program texts.

Navigation based on the text cursor position LSP defines no protocol that notifies movement of the text cursor, but several protocols (e.g., `Hover Request` and `Code Lens Request`) that request additional information related to the editing code. We might be exploiting those notifications to detect the movement of the cursor position. Otherwise, we would need to define our own protocol to notify movement of the text cursor as it is crucial to implement Kanon's feature.

Controlling the text cursor position LSP defines no protocol that controls the text cursor position from the server's side. Therefore, Kanon's features that move the cursor position by clicking visualization cannot be implemented straightforwardly. One approach would be to install a new command into an editor, which can be triggered by the server.

5 Conclusion

This paper proposed *Poly²Kanon*, an implementation framework of live data structure programming for multiple languages and environments. The core ideas of the framework are to use a meta-JIT language implementation framework for minimizing dependency on the target language, and to use the Language Server Protocol (LSP) for separating the editor implementation from the live programming engine.

We investigated the meta-JIT language implementation framework and LSP with respect to the Kanon's implementation details and features, and presented a rough sketch of the *Poly²Kanon* API for supporting multiple languages. For supporting multiple environments, we presume that the current LSP might not be sufficient and that further investigation is required.

Acknowledgement

Niephaus et al. developed a live programming system by using Graal/Truffle and LSP [8]. The core part of this work was proposed independently from their work. We would like to Fabio Niephaus for many suggestions on a draft version of the paper. We would like to thank the reviewers of the LIVE 2020 workshop for their valuable comments. This work was supported by JSPS KAKENHI grant number 20K21790.

References

[1] Aaron, S., and Blackwell, A. F. From Sonic Pi to Overtone: Creative musical experiences with domain-specific and functional languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design* (New York, NY, USA, 2013), FARM '13, ACM, pp. 35–46.

[2] Bolz, C. F., Cuni, A., Fijalkowski, M., and Rigo, A. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (New York, NY, USA, 2009), ICOOOLPS '09, ACM, pp. 18–25.

[3] Jakobs, F. Ace: The high performance code editor for the web. https://ace.c9.io, 2018. Accessed on April 23, 2018.

[4] Khan Academy. Intro to JS: Drawing & animation. https://www.khanacademy.org/computing/computer-programming/programming, 9999. Accessed Ferburary 2017.

[5] Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. The Scratch programming language and environment. *ACM Transactions on Computing Education (TOCE) 10*, 4 (2010), 1–15.

[6] McDirmid, S. Living it up with a live programming language. In *In Proceedings of Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (2007), ACM, pp. 623–638.

[7] Microsoft. Language server protocol. https://microsoft.github.io/language-server-protocol/. visited September 19, 2020.

[8] Niephaus, F., Rein, P., Edding, J., Hering, J., König, B., Opahle, K., Scordialo, N., and Hirschfeld, R. Example-based live programming for everyone: Building language-agnostic tools for live programming with LSP and GraalVM. In *Proceedings of the ACM Symposium for New Ideas, New Paradigms, and Reflections on Everything to do with Programming and Software (Onward! 2020)* (Nov. 2020).

[9] o. DeLine, a. Fisher, a. Chandramouli, o. Goldstein, i. Barnett, a. Terwilliger, and o. Wernsing. Tempe: Live scripting for live data. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), pp. 137–141.

[10] Oka, A., Masuhara, H., and Aotani, T. Live, synchronized, and mental map preserving visualization for data structure programming. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (New York, NY, USA, Nov. 2018), Onward! 2018, ACM, pp. 72–87.

[11] Post, E. Real programmers don't use Pascal. *Datamation 29*, 7 (1983), 263–5.

[12] Rein, P., Taeumel, M., Hirschfeld, R., and Perscheid, M. Exploratory development of data-intensive applications: Sampling and streaming of large data sets in live programming environments. In *Companion to the First International Conference on the Art, Science and Engineering of Programming* (New York, NY, USA, 2017), Programming '17, Association for Computing Machinery.

[13] Swift, B., Sorensen, A., Gardner, H., Davis, P., and Decyk, V. Live programming in scientific simulation. *Supercomputing Frontiers and Innovations 2*, 4 (2016).

[14] Tanimoto, S. L. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)* (2013), IEEE, pp. 31–34.

[15] Würthinger, T., Wöss, A., Stadler, L., Duboscq, G., Simon, D., and Wimmer, C. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (New York, NY, USA, 2012), DLS '12, ACM, pp. 73–82.