Co-evolution as the Key for Live Programming

Remo Lemma and Michele Lanza
REVEAL @ Faculty of Informatics – University of Lugano, Switzerland

Abstract—**The promise of live programming is to shorten or even break the infamous edit-compile-run cycle, providing live feedback on a program's envisioned behavior while it is being written. Several live programming languages and environments exist, from venerable examples (Smalltalk, LISP) to more recent efforts like Ruby. In most cases either the IDE comes as an afterthought, after the language is designed, or novel languages are made to fit into existing IDEs. We pursue a middle ground by co-evolving both a language and its IDE: we are developing a novel live programming language, called Moon, from scratch, and are concurrently building its IDE. We illustrate our efforts so far and discuss our overall vision.**

I. Introduction

When developing software systems, modeling and other activities heavily influence the final outcome, but primarily programming moves abstract thoughts to concrete artifacts. Most mainstream programming languages (and environments) are based on the traditional edit-compile-run cycle. This approach allows developers to recognize clear boundaries between the different phases to focus on one activity at a time: first write the code, then compile it, and finally observe and test the system at runtime. Although this approach is easy to understand and use, it is sub-optimal. While writing code, developers have problems understanding the impact of modifications and the frequent interruptions, due to (re-)compilation and program startup, have a negative effect on the productivity of developers.

Live programming systems provide immediate and live feedback on a program's runtime behavior while the code is being changed. This cuts waiting times and ensures that developers are aware of the changes made to the running system. In contrast to live programming precursors like Smalltalk and LISP, modern live coding is often based on the use of audiovisual artifacts, allowing one to smoothly work on both sides: For example it is possible to draw shapes to generate code, and it is possible to write code to generate shapes.

Adapting existing languages and/or environments to the live programming philosophy is non-trivial, especially when they do not natively support introspection and reflection. The creation of new solutions is often approached by developing languages and environments separately. A common approach (*e.g.,* [1]) is to develop a new language and afterwards exploit an existing IDE (*e.g.,* Eclipse). This does not come without technical constraints, and concessions have to be made to make language and IDE co-exist. We believe that existing IDEs are not suitable for live programming, as they have not been designed for this.

We present our ideas about a novel live programming language —named Moon—, and its development environment, which we are currently building in parallel.

II. Moon: Inception & Rationale

In designing Moon we take a bottom-up minimalistic approach, starting from few essential elements. We started by implementing the primitive numeric types and the basic mathematical operations. We then added the possibility to create user-defined functions. This allows us to create behavioral entities which can refer to each other and execute simple mathematical operations. Concurrently we started developing a dedicated IDE with native support for live feedback. We developed two prototypes, depicted in Figure 1:

(A) The first one is directly integrated in Pharo[1], an open source Smalltalk implementation which we used to create the first version of the compiler for Moon.
(B) The second prototype is a web-based IDE disconnected from the compiler, written in JavaScript. It uses HTML5 for the front-end.

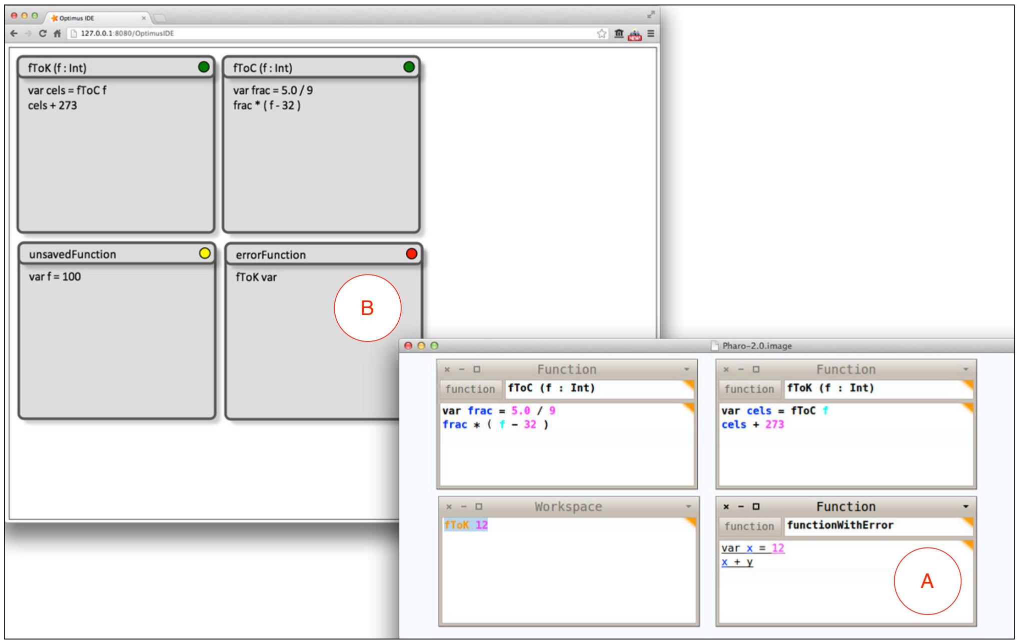
We envision an environment which immediately reacts and provides visual feedback (we also plan for aural feedback) on the individual entities, on the state of the system and on its evolution. In the following we discuss our current ideas.

Entities Visualization. When visualizing the state of a component (*e.g.,* an instance of a class, the result of a program) what matters is its nature. Live programming is perfectly suitable for audiovisual domains, yet we believe that it is also applicable in more abstract settings. We want to base our environment (and language) on the concept of *Representation*. Each entity is associated with its concretization in a human-perceptible format (*i.e.,* a *Representation*). Basic representations will be shipped with the system, and users will be able to compose them in order to create new representations for their custom entities. This allows us to abstract away from the concept of audiovisual feedback, and empower users with the ability to choose appropriate ways to represent a certain concept, mixing, if necessary, different types of feedback.

State Visualization. Having live feedback on the overall state of a system is valuable, as it allows developers to spot early errors, if they see the impact of their actions not only on the current entity, but also on the overall system.

We implemented (consult Figure 1) a first prototype of state visualization. In Pharo we underlined parts of code which do not compile (and the compiler is invoked at every carriage return). In the web-based environment we use colors to distinguish the state of the different parts of the system (*e.g.,* green for correctly compiled, yellow for modified, red for erroneous), as we did not implement live syntax highlighting yet and compilation has to be triggered manually.

[1]http://www.pharo-project.org/



```
fToK (f : Int)
var cels = fToC f
cels + 273
```

```
fToC (f : Int)
var frac = 5.0 / 9
frac * ( f - 32 )
```

```
unsavedFunction
var f = 100
```

```
errorFunction
fToK var
```

```
Function
function fToC (f : Int)
var frac = 5.0 / 9
frac ∗ ( f − 32 )
```

```
Function
function fToK (f : Int)
var cels = fToC f
cels + 273
```

```
Workspace
fToK 12
```

```
Function
function functionWithError
var x = 12
x + y
```

Fig. 1: Moon code written in the two versions of the development environment.

The current approach can be greatly improved, by providing automatic compilation at every change in the abstract syntax tree, and by using more sophisticated visualizations to show the state of the system. We advocate the use of an overlay in which the negative impact of the last change is shown. For example if the last change triggers an exception in another function the overlay will show these dependencies, which can also be conceived as a *visual debugging* technique.

Evolution Visualization. While working on a system, especially in a collaborative settings, we can exploit the concept of live feedback to keep always an updated visualization of the overall evolution of the system. We are still at the inception of this idea, but we believe it is valuable to be explored for future research. As a first try we will tackle this problem by creation an evolving map [2] of the software at hand.

III. Conclusions & Future Thoughts

Our goal is to co-evolve a live programming language and its environment. Novel IDEs (*e.g.,* Gaucho [3]) have been leveraged in the context of new metaphors to facilitate program comprehension and modern languages (*e.g.,* Scala[2]) are equipped with newer, higher-level, abstractions which ease programming. However, when it comes to live programming, we doubt that the union of these two distinct worlds can produce a satisfactory outcome. The development of a custom solution which fully integrates the philosophy of live coding is essential. Although ad-hoc environments for specific languages have been created in the past (*e.g.,* [4]), co-evolving them gives the advantage of having the ability to test in an incremental way how users react to the whole (language plus environment).

[2]http://www.scala-lang.org

We plan to integrate the ideas discussed above in Moon and in its development environment. Because we plan to integrate the *Representation* based approach, we also aim at using it to test its usability on the language itself (*i.e.,* by representing the entities composing the language, which are in turn code entities, thus, representable). However, there are a number of issues that have to be tackled and that we will investigate:

- How can we integrate the possibility to write parallel and/or multi-threaded source code and what is the best way to give live feedback on its execution (and on possible bugs and/or errors)?
- What is the level of liveness that the system should have? In a Smalltalk-like philosophy the whole system should be always live and also shipped as-is. However this is not always the best choice (*e.g.,* for performance-sensitive applications). On the other side, deploying full-blown, compiled, applications would be detrimental for those cases where having a live system could be beneficial.

References

[1] S. McDirmid, “Living it up with a live programming language,” in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, 2007, pp. 623–638.
[2] A. Kuhn, D. Erni, P. Loretan, and O. Nierstrasz, “Software cartography: thematic software visualization with consistent layout,” *Journal of Software Maintenance*, vol. 22, no. 3, pp. 191–210, 2010.
[3] F. Olivero, M. Lanza, and M. Lungu, “Gaucho: From integrated development environments to direct manipulation environments,” in *Proceedings of FlexiTools 2010 (1st International Workshop on Flexible Modeling Tools)*, 2010.
[4] R. B. Smith, J. Maloney, and D. Ungar, “The Self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility,” in *OOPSLA ’95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*. ACM Request Permissions, Oct. 1995.