Liveness becomes Entelechy - A scheme for L6

Luke Church
Google Inc.
luke@church.name

Emma Söderberg
Google Inc.
emso@google.com

Gilad Bracha
Google Inc.
gilad@bracha.org

Steven Tanimoto
University of Washington
tanimoto@cs.washington.edu

ABSTRACT

Integrated development environments have provided increasingly powerful tools for software creation, and yet the creation of complex computer programs remains difficult and time-consuming. Liveness in a programming environment has been identified as one direction in which to pursue further improvements in programmer productivity. We propose a scheme for achieving strategically predictive liveness, that is a scheme which can predict and evaluate considerable features of an application. The scheme exploits statistical properties of code to allow for synthesis and evaluation of code that is most likely to be useful to the developer. We hypothesise that this will help inculcate liveness into mainstream technical practice.

1 Introduction

Tanimoto (1990) introduced a taxonomy of levels of liveness describing the interactive cycle that a user has with a system. In the context of programming environments, a program representation-plus-environment can often be put into one of four categories: (L1) useful to a human but not a computer (e.g., hand-written flowchart on paper), (L2) executable, and run on request, (L3) automatically re-run after the completion of a meaningful edit operation, such as the replacement of a statement or expression, and (L4) fully live, in which the program can be edited and code sections replaced without stopping the running execution.

Tanimoto (2013) subsequently extended this taxonomy shifting the focus to contexts where the computer no longer simply reacts, but also anticipates the needs of the programmer. In (L5) the computer is tactically predictive and suggests small changes to the program and evaluates them, in (L6) the computer becomes strategically predictive inferring large aspects of functionality and executing them.

The trend over the years has been to see more and more systems in levels 3 and 4, including systems built by the authors. In this paper we propose a mechanism to allow the user to select programs and the results of those programs that their future self would have written, closer to an L6 system. As such, liveness becomes the motive force of programming systems, its entelechy.

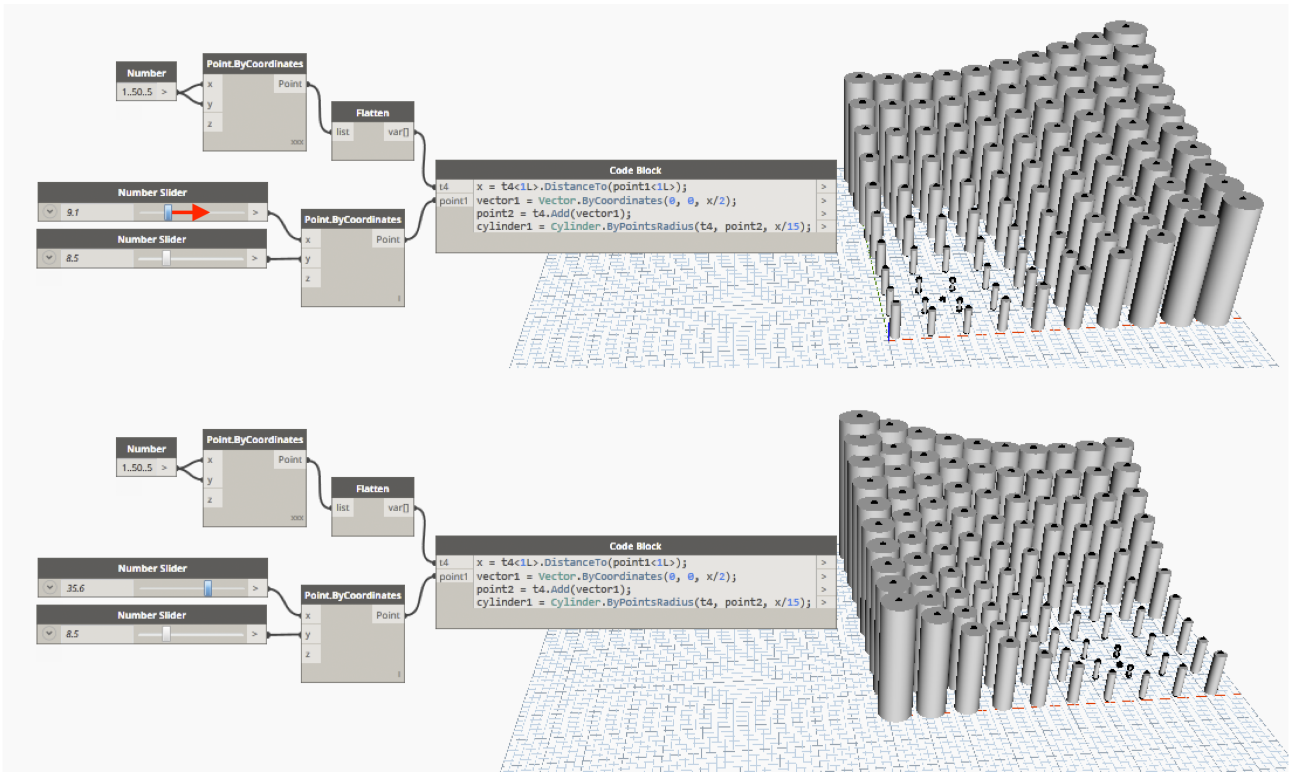
Let's start by reviewing programming systems available at L3 and L4.

1.1 Liveness L4: Dynamo

Visual languages using dataflow frequently support automatic execution and adaptation in response to a stream of edits. For example in the ("Dynamo" 2016) graph below, when the user slides the slider (indicated by the red arrow) the Dynamo virtual machine computes the change set of the nodes that need to be re-evaluated and updates the display (the second image below) to reflect the new state. This means that there is an immediate L4 streaming feedback cycle between the interaction with the UI components and the result.

Dataflow execution makes it comparatively easy to compute the change set necessary to respond to a change. However, whilst liveness is a natural property of such systems, it is not only a property of data-flow systems. Other languages have supported similar properties for a long time, using different implementation mechanisms.

For example, there is a rich tradition of using live programming systems for the generation and performance of music (A. Blackwell and Collins 2005) and ("TOPLAP" 2016). Examples of current systems include Sonic Pi (Aaron and Blackwell 2013) and Tidal (McLean 2014). Many of these systems operate at L3, with the musician updating the program text and using a shortcut key to update the running program. These interactions often occur at such speed that it appears to the audience as a stream of edits.



```
x = t4<1L>.DistanceTo(point1<1L>);
vector1 = Vector.ByCoordinates(0, 0, x/2);
point2 = t4.Add(vector1);
cylinder1 = Cylinder.ByPointsRadius(t4, point2, x/15);
```

```
x = t4<1L>.DistanceTo(point1<1L>);
vector1 = Vector.ByCoordinates(0, 0, x/2);
point2 = t4.Add(vector1);
cylinder1 = Cylinder.ByPointsRadius(t4, point2, x/15);
```

Figure 1: *Dynamo, an L4 live programming system*

1.2 Liveness L3/L4: Smalltalk and Newspeak

Languages in the Smalltalk family have long supported liveness at L4, though tools usually manifest this only at L3 (i.e. they require a manual step to update the program). One can add, remove or modify methods and fields at runtime, or change the class hierarchy. If a field of a class C is added or removed, all instances of C (and its subclasses) are immediately reshaped accordingly. The same holds if a class' superclass is reshaped, directly or indirectly.

Changing methods in a running program poses challenges. The method in question might have activations on the stack. In these cases, all such activations (and any activations above them) are popped, and execution resumes at the first call to the altered method. Similar facilities are being developed as part of the ("Dartino" 2016) Dart VM.

While the reflective API allows program changes to be invoked from within the program at level 4, interaction with the programmer is typically at level 3. Programmers make changes while the program is halted and then resume execution. Newspeak (Bracha et al. 2010) has started to introduce limited level 4 interaction (Bracha 2016)

A fair amount of development in Smalltalk still happens via editing of dead code (i.e. L2), in class browsers. In Newspeak, we are experimenting with more pervasive liveness wherein all code is always displayed in the context of live program data (Bracha 2013).

Infrastructure with these kinds of capabilities is the starting point for our work.

2 Speculative execution: Liveness L5

Looking at how we can extend Liveness beyond this, let's start by considering a system for achieving L5 liveness, that is one that is tactically predictive. Here we need a system that will be able to guess what function the developer was about to type and run that. There are a number of challenges in doing this; computational cost, parameter satisfaction, side-effects and visualization. We'll tackle them in order.

2.1 Computational Cost

The principal problem with speculative execution is the extreme computational complexity. Considering a very naive example of potential programs of 38 characters. One possibility is '`main() { print (new DateTime.now()); }`', another is '`main() { var a = 42; print (a - 42); }`' and a third is '`IAmAMarmot.IAmAMarmot.IAmAMarmot.IAmAM`'.

Simply attempting to execute all programs of a given length is computationally absurd, even under conservative assumptions there are some approximately 10^{67} programs of 38 characters.

However as the above example hints at, not all of these sequences are equally likely to be what the developer meant. As we described[1], we can use a corpus of source code and a bayesian model to compute an ordering on the likelihood of the various choices for a given call site.

By executing methods in this order (using hot-patching to apply the change to the previous state of the VM) we can spend our cpu time so that the code the developer would like to write and its associated results have probably been computed by the time the developer is ready to consider them.

It is also worth noting that computer time is often vastly cheaper than people's time. For $50/hour we can rent 1000 cores with 3.7TB of RAM[2]. This means that we can deploy very considerable computational resource if we can make use of a distributed architecture to improve developers' productivity. We can expect the available optimisation power to improve over time. The question then becomes how can we use this extensive resource to assist people writing code?

2.2 Parametric Satisfaction

A second problem is that many function calls need arguments. Take for example the function math.cos(x); What is the argument, x, that will be passed to cos? Similarly to the case above, we can build a probability distribution using various factors to prioritise the order of evaluation. For example, it is more likely that x will be a number than a string, it is more likely that it will be a variable that is in scope, etc. These can all be computed as a distribution over the corpus of programs. The same strategy can be used to unify the model of the target of a method call compared to a static function call (e.g. is `i.abs()` more likely than `math.cos(i)`)?.

To give an indication of the power of this technique, we can look at the distribution of the cos function over a sample of n = 100,000 call sites from a corpus of public Dart code on GitHub.

Fraction	Description
0.1%	constants (e.g. `cos(4);`)
82.6%	variable accesses (e.g. `cos(angle);`)
7.9%	expressions of depth 2 (e.g. `cos(angle / 2);`)
4.6%	expressions of depth 3 (e.g. `cos(angle / (2 * PI));`)
4.8%	expressions of depth 4 or more

This relatively shallow depth of the expression, means that we can achieve very good coverage of probable code by considering only expressions of depth 3 or fewer.

The other advantage of the design here is that we have a live system. Consequently it is simple to stop the program at the execution site and enumerate all the possible objects that are in scope, these making up the list of viable variable accesses.

2.3 Side-effects

The combination of the analysis above gives us a way of selecting functions and lacing them together into a scheme that allows us to try executing them to see what the results are, selecting them in a reasonable order of probability. From here, there are two harder problems to deal with, the first is so called 'side-effect' management.

Consider the function call `File.Delete("/", recursive: true)`. Speculatively executing this may have unwanted effects besides showing to users what the results might be. In order to understand this better, we can separate the effects of calling functions into different groups: effect-free, transient effects and persistent effects.

Effect-free functions are functions where there is no externally observable effect other than the result. For example the List.length getter. These functions may always be executed safely and without concern.

The second form is transient effects. These are effects that are observable only within the virtual machine. We propose to use Warth et al. (2011)'s worlds scheme to facilitate easy garbage collectable experiments with the addition of a mechanism to compare the two worlds, this will allow us to speculatively execute functions comparing their results.

[1]https://lang-games.blogspot.com/2016/03/simple-code-completion-ordering.html
[2]https://cloud.google.com/compute/pricing current as of March 2016

The third form is persistent effects, these are effects that are observable from outside the virtual machine, e.g. on the file system or the network. These could be managed by using a sandboxed ‘universe’, but for now, we propose the simpler scheme of preventing these methods from speculatively executing without explicit permission from the user.

This three layer scheme gives us a way of most efficiently executing and undoing the execution of most functions. We annotate the core library with labels denoting which, progressively more expensive, scheme should be applied for undoing the effects of a function call. We can then propagate these labels using a most-conservative-wins strategy to transitively compute the label that an arbitrary function should have. There are a wide variety of other techniques available should these prove insufficent, for example Barr and Marron (2014)’s TARDIS system or Elm’s time travelling debugger Czaplicki and Chong (2013).

Combining this with the above, we have a scheme which we can use to speculatively evaluate small increments on a program, in approximate order of their likelihood of being useful.

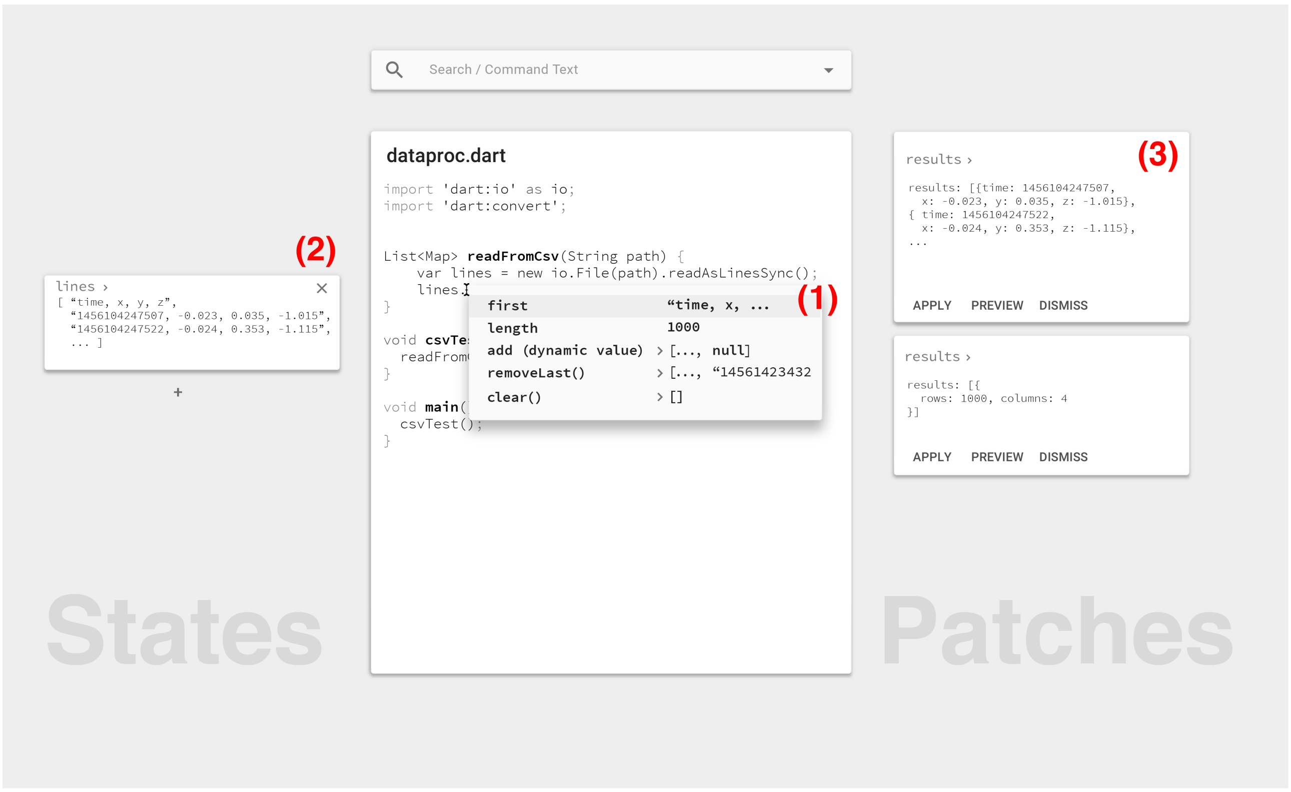
This leaves two challenges, one is how to support the developer in selecting which option to execute, and the second is growing the size of the predictions.

2.4 Visualisation: Selecting between results

Liveness offers a fundamental improvement in developer user experience: the user can work primarily in the domain of the results of programs, rather than between the programs themselves. With an L3/4 system they can rapidly experiment with potential programs and see the results immediately, within L5/L6 we can take this further and have the user consider different outcomes and select the one they want. This has a relationship to programming by example systems that we’ll explore later, for now we note that this places a different pressure on the design of the user interface, that is it becomes an interface where the current and future states of the program are paramount rather than the code. There is some related research in this space in design disciplines (Bradner, Iorio, and Davis 2014)

Within physical design the notion of what is the output is often comparatively easy to know; the user selects the physical artefact they are interested in. In programming we don’t know a priori what are the important artefacts. We address this by focussing on visualising the change from the base program.

Below we present a work in progress as to how such a user interface might look.



```
import 'dart:io' as io;
import 'dart:convert';


List<Map> readFromCsv(String path) {
    var lines = new io.File(path).readAsLinesSync();
    lines.
}

void csvTe
  readFrom
}

void main(
  csvTest();
}
```

```
first                  “time, x, ...
length                 1000
add (dynamic value)  › [..., null]
removeLast()         › [..., “14561423432
clear()              › []
```

```
[ “time, x, y, z”,
  “1456104247507, -0.023, 0.035, -1.015”,
  “1456104247522, -0.024, 0.353, -1.115”,
  ... ]
```

```
results: [{time: 1456104247507,
  x: -0.023, y: 0.035, z: -1.015},
{ time: 1456104247522,
  x: -0.024, y: 0.353, z: -1.115},
...
```

```
results: [{
  rows: 1000, columns: 4
}]
```

Figure 2: *Work in progress prototype for selecting between results of programmmes*

The user writes the beginning of the source code, the code completion dialog (1) is ordered by the the likelihood of selection, the right hand side shows the results of selecting that call. This gives an L5 interaction style. (2) shows the states of the variables that are likely to be of most use at the specific location. (3) shows the result larger structural possibilities, adding features to the software. Selecting the card containing the proposed change will adjust the contents of the source panel with the source changes whose execution resulted in the values shown in (3).

3 Strategic prediction: Liveness L6

We have now outlined a scheme that allows us to effectively operate at L5, the remaining challenge is to increase the scope of the predictions such that larger chunks of functionality can be predicted. This is the area of most active research that we are exploring.

To give a first example, consider adding logging to a Dart application, this requires a couple of components, the addition of the log calls and the registration of the log handler. The number of different implementations of log handlers tends to be fairly small, mainly applying some formatting and delivering the messages to an output target, such as stdout. By modelling the differences between various implementations and surfacing them as either user selections or parameters we can build a form of statistical abstraction, which can be predicted as a whole.

The second observation is that the way in which such objects tend to be used is highly patterned, encoded in ‘object protocols’ (Beckman, Kim, and Aldrich 2011). For example, a file might be always opened, read, and closed. We can use these to predict usage at the call sites.

Taken together this allows the usage of clusters of objects to be predicted as a whole. Once we have a scheme like this of how an API is being used, we can look for structural analogies between the usage of the object patterns and other libraries, offering the potential to be able to map from libraries in other languages.

3.1 Plausibility: Introducing Small Statistical Dart (SSDart)

How plausible is all this? To investigate we’re implementing a concept prototype, SSDart, which operates over a small fraction of the Dart language (construction, assignment, method invocation) and tiny subset of the APIs (lists, string manipulation, file). The statistical models are relatively crude, but still much better than random.

The work is at an early state. Our prototype SSDart engine already evaluates 1000 possible dart programs per second. Combined with the simplistic Bayesian model for ordering code completion ordering prediction is sufficient to assist the developer in over 90% of completions.

This gives us a platform for performing more sophisticated experiments.

4 Further work

There are four primary areas of work that need more development: runtime and static data fusion, programming by example, smarter candidate generation, and improving selection mechanisms.

The runtime behaviour of programs carries information as to whether it was the programme the developer intended to write. For example, if executing a fragment results in, for example, a null reference exception, this makes it less likely that this is the code the user wanted to write. There is further work to be done building a systematic way of fusing the information gained from live executing a program with the information derived from analysing a corpus of code.

A related area is integrating direct data from the user. Consider the example above of processing the csv file, if the user supplied a sample of what they would like the outcome of a function to be, this could be used as a strong prior for selecting the functions.

A third area requiring further work is the candidate generation functions. The usability of the system is dependant on being able to deliver a stream of possible, probable, programs to the user. This depends on being able to generate a stream of such programs to evaluate. The algorithm used in SSDart is a naive breadth first search with a depth limitation.

A final area of improvements is the user interface for selecting the options. Being able to fluidly and interactively evaluate and select between these options is crucial for the user experience.

5 Conclusion: Liveness becomes Entelechy

Liveness at L3 or L4 will soon be conventional in programming languages. We have seen how by combining these liveness properties with a statistical model of programs allows us to build an interaction paradigm of selecting between potential future functionalities.

The liveness of the system is the essential motive force that provides the ability to consider these future possible programs' results, or its entelechy.

6 Acknowledgements

The authors would like to thank Alan Blackwell, Kai Backman, Erik Ernst, Thomas Green, Elizabeth Kemp, Mariana Marasoiu and the anonymous reviewers for valuable feedback.

References

Aaron, Samuel, and Alan F Blackwell. 2013. "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*, 35–46. ACM.

Barr, Earl T., and Mark Marron. 2014. "Tardis: Affordable Time-Travel Debugging in Managed Runtimes." In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 67–82. OOPSLA '14. New York, NY, USA: ACM. doi:10.1145/2660193.2660209.

Beckman, Nels E, Duri Kim, and Jonathan Aldrich. 2011. "An Empirical Study of Object Protocols in the Wild." In *ECOOP 2011–Object-Oriented Programming*, 2–26. Springer.

Blackwell, Alan, and Nick Collins. 2005. "The Programming Language as a Musical Instrument." In *Proceedings of PPIG05 (Psychology of Programming Interest Group)*.

Bracha, Gilad. 2013. "Making Methods Live." http://gbracha.blogspot.co.uk/2013/04/making-methods-live.html.

———. 2016. "Newspeak by example." http://bracha.org/Literate/literate.html.

Bracha, Gilad, Peter Von Der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. "Modules as Objects in Newspeak." In *ECOOP 2010–Object-Oriented Programming*, 405–28. Springer.

Bradner, Erin, Francesco Iorio, and Mark Davis. 2014. "Parameters Tell the Design Story: Ideation and Abstraction in Design Optimization." In *Proceedings of the Symposium on Simulation for Architecture & Urban Design*, 26. Society for Computer Simulation International.

Czaplicki, Evan, and Stephen Chong. 2013. "Asynchronous Functional Reactive Programming for GUIs." In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 411–22. PLDI '13. New York, NY, USA: ACM.

"Dartino." 2016. http://dartino.org/.

"Dynamo." 2016. http://dynamobim.org/.

McLean, Alex. 2014. "Making Programming Languages to Dance to: Live Coding with Tidal." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, 63–70. ACM.

Tanimoto, Steven L. 1990. "VIVA: A Visual Language for Image Processing." *Journal of Visual Languages & Computing* 1 (2). Elsevier: 127–39.

———. 2013. "A Perspective on the Evolution of Live Programming." In *Live Programming (LIVE), 2013 1st International Workshop on*, 31–34. IEEE.

"TOPLAP." 2016. http://toplap.org/.

Warth, Alessandro, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. 2011. "Worlds: Controlling the Scope of Side Effects." In *ECOOP 2011–Object-Oriented Programming*, 179–203. Springer.