Live Literals

Tijs van der Storm, Felienne Hermans

Live programming environments improve programmer experience by providing views of program execution which are continuously, and instantaneously updated. In most existing work on liveness, these views are considered part of the IDE: separate windows, panels, or widgets allow programmers to inspect and interact with live data and program execution. In this paper we present "live literals" where the source code itself is used as vehicle for immediate feedback and direct manipulation. Live literals are like ordinary programming language literals, but they are automatically updated after changes to the code. We illustrate the concept of live literals in Javascript using three applications: embedded spreadsheets, live units tests, and probes.

Introduction

The goal of live programming is to improve programmer experience through better feedback and better tools to manipulate code. Often, the responsibility for these additional affordances is on the IDE: the user interface, for example, provides separate panels and widgets for inspecting and manipulating program code and execution. The source code itself is more or less taken for granted as being a completely static input artifact. In this work we make a first step towards turning this hierarchy upside-down. Instead of proposing further adornments of the IDE, we propose to make the source code itself more dynamic and interactive.

We introduce *live literals*: literal data expressions in the source code which are updated continuously while the code is executed or in reaction to program edits by the programmer. The archetypical example of live literals is captured in the following snippet of code: `doIt("1 + 2", 3)`. Whenever the programmer edits the first argument, the second argument is replaced with a literal rendering of the result of evaluating the quoted code. For instance, changing `"1 + 2"` to `"1 * 2"`, changes the constant `3` to `2`. Live literals thus allow the programmer to directly manipulate[1] input data from within the code, as well as receive immediate feedback about their actions, also within the code[2].

With live literals, the programmer may enjoy immediate feedback right from the place where it is most relevant, without having to switch to different windows or panels. Furthermore, because live literals are really just source code, the provided input and computed feedback can be copy-pasted, stored, versioned, and shared at will. Finally, since live literals live inside ordinary statements and expressions, they play well with ordinary programming abstractions, such as loops, conditionals and function abstraction.

We illustrate live literals using three applications implemented in Javascript. The first allows functions to be annotated with example data to explore and test the result of evaluating a function. The second exploits Javascript's object literal syntax to represent spreadsheets in ordinary Javascript code. Finally, we show how live literals can be used to implement *probes*, which are basically print-statements on steroids. The paper is concluded by briefly reflecting on implementation concerns and sketching an outlook for further work.

[1] B. Shneiderman, Direct Manipulation: A Step Beyond Programming Languages, *Computer*, Volume 16 , Issue 8, 1983.

[2] Henry Lieberman and Christopher Fry, Bridging the gulf between code and behavior in programming, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'95)*, pp. 480–486, 1995.

Live Literals: Syntactic Real-Estate for Direct Manipulation and Immediate Feedback

Literal expressions exist in all programming languages, from the atomic integer, string and boolean literals, to composite literals for representing lists (Lisp), arrays (Ruby, Python, etc.), objects (Javascript, Newtonscript), or XML documents (Scala). Normally, literals represent static aspects of data, for instance to initialize parts of a computation. They are static, because their layout (or value) is determined even before compiling the code, and the code itself is not assumed to change at run time.

Live literals are different. Instead of representing static inputs to some computation, live literals are input and output ports for dynamic data, *at edit time*. Editing a literal may trigger a computation, the result of which will be instantaneously fed back into the source code, at a different location. One literal's input feeds back into another literal as output. As a result, instead of a serial script for the computer to execute, the text itself becomes as interactive as a user interface.

Motivation

But why would this be relevant or even useful? A first motivation for live literals is simply "less is more". Live literals are part of the primary artifact of the working programmer, the source code. There is no need for separate panels, views or popups, which divert the programmer's attention from the actual source code. Consequently, live literals promise a significantly simpler model of the IDE as a whole, since part of the traditional IDE services are taken over by the source code itself. Finally, the representation through data literals entails that the language of feedback is the same as the language of code, which again simplifies things considerably: one language is simpler to deal with than multiple languages. We conjecture that live literals improve feedback, paying the minimum cost in terms of cognitive distance and representation impedance mismatch, virtually eliminating all context switching.

Another benefit of live literals is that reifying input and output data as part of the source code allows this information to be shared. For instance, just like ordinary code, code with live literals can be copy-pasted, persisted on disk, and versioned using a configuration management tool. This will be helpful in (online) collaboration settings, where collaborators do not have to follow elaborate instructions to reproduce certain situations within their own workspace.

As a final potential benefit of live literals as opposed to separate IDE views is that they are amenable to standard programming abstractions. Since live literals are part of ordinary expressions and statements, they can partake in conditional, looping control-flow, or functional abstraction. This enables patterns of use that are not anticipated by the canned visualizations offered in traditional IDEs. Although we have not fully explored this new power, we expect this to provide an additional level of flexibility for improving how we interact with our source code.

Exploring Live Literals: What You Program is What You See (WYPIWYS)

Below we discuss three applications of live literals in Javascript: live examples and tests, embedded spreadsheets, and probes.

The editors give special meaning to certain Javascript function calls containing data literals, and then change the source text of those literals depending on the application. Note that the code is ordinary Javascript: without the live literal language support, the code just runs fine, and the special cased function calls simply compute valid results or gracefully degrade into no-ops. The examples are all live, so the reader is encouraged to tinker with the source code and observe what happens.

Examples and Tests

The first application of live literals allows the programmer to explore the behavior of a function definition using live data and live tests, similar to Gilad Bracha's live methods[3]. The code below shows a function to compute the factors of integers.

[3] Gilad Bracha, Making Methods Live, Room 101, April, 2013.

Live examples and tests. Change the value
:eral
:ry is

ates

```
function factors(n) {
  test([
    {n: 0, result: []},
    {n: 4, should: [1,4,2], result: true}
  ]);
  var fs = [];
  for (var i = 1; i <= Math.floor(Math.sqrt(n)); i++)
    if (n % i === 0) {
      fs.push(i);
      if (n / i !== i) fs.push(n / i);
    }
  return fs;
}
```

The `test` invocation at the beginning describes a fixture for executing `factors`. The fixture is a plain Javascript array containing object literals specifying input and output of executing `factors`. For instance, the first object literal represents the base case where `n` is zero. The result field shows that zero does not have any factors. Try changing the constant 0 to another number and observe that the result component changes. The second literal represents a test. In this case, the user provides a `should` entry, containing the expected result. If this component is present, the test interpreter checks the result of evaluating the function against the expected result, and indicates success or failure of the test in the `result` entry. Again, this result is updated whenever the input or expected output literals are modified.

Embedding Spreadsheets

Spreadsheets are well-known end-user programming environments for managing tabular data and simple computations. One advantage of spreadsheets is their liveness: the grid-based UI supports direct manipulation of input data and code expressions, and the effect of changes is immediately reflected in the UI itself. Live literals can support the model of computation, but this time directly within the source code instead of a grid-based UI.

The code below shows a simple spreadsheet for calculating grade averages and the class average based on lab and exam grades for students.

```
cell(function avg(sheet, lab, exam) {
  return (lab + exam) / 2;
});

cell(function classAvg(sheet) {
  return avg('avg', sheet);
});

var grades = sheet([
 {classAvg: 8.375},
 {student: 'James', lab: 9, exam: 9, avg: 9},
 {student: 'Sean', lab: 8.5, exam: 7, avg: 7.75}
]);
```

Embedded spreadsheets in code. The grades

g
the
heet.

The first two statements define two cell functions using the `cell` function which is provided by our framework. The first one defines the average of the lab and exam grades. The second one computes the class average over the complete sheet. The library function `avg` computes the average over a named column (in this case 'avg'). Both cell functions get the complete sheet as the first argument to allow arbitrary computations over a sheet.

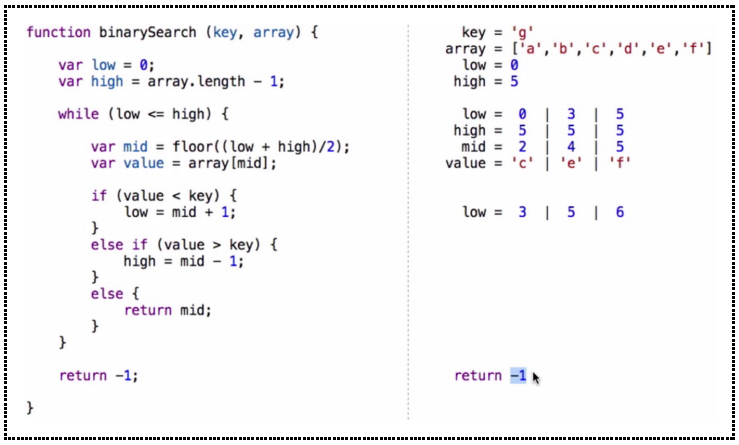
The variable `grades` is initialized with a *sheet literal*, indicated by the `sheet` function. The sheet literal consists of list of rows, containing object literals with named cells. The `sheet` function evaluates the sheet literals by applying cell functions where needed, and returns the resulting object. For every row the available data cells (`lab` and `exam`) are matched to the formal parameters of the cell functions, and if there is a match, the computed cell (named after the cell function) is inserted or updated in that row. For instance, the last two rows of the sheet literal contain the `avg` cell, since both these rows contain `lab` and `exam`, as required by the cell function `avg`. The first row, however, does not have these input cells, so does not get the `avg` cell. In the background, the `sheet` function computes all dependencies between cells and evaluates the computed cells. The result is then piped back into the editor to reflect the actual state of the sheet. The result is that sheet literals provide the same live behavior as traditional spreadsheet systems, only the user interface is completely textual.

Probes

Our final application of live literals is inspired by Bret Victor's influential talk *Inventing on Principle* and Sean McDirmid's work on probes[4]. The following screen capture of Victor's talk shows two panels. The left shows the code of a function which implements binary search. The right shows the actual values of the local variables of the function while it executes.

[4] Sean McDirmid, Usable Live Programming, *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward!'13)*, pp. 53–62, 2013.

From Bret Victor, Inventing on Principle, 23th minute, CUSEC, 2012



Our implementation of probes eliminates the two views, and integrates the display of the runtime values of variables into the source text itself. The following code shows the same binary search function, enhanced with probe statements, indicated by function calls to `p`.

```
function binarySearch(key, array) {
    run({key: 'g', array: ['a', 'b', 'c', 'd', 'e', 'f']});

    var low = 0;
    var high = array.length - 1;

    while (low <= high) {
        p(low, [0,3,5]);
        p(high, [5,5,5]);

        var mid = Math.floor((low + high)/2);
        var value = array[mid];

        p(mid, [2,4,5]);
        p(value, ["c","e","f"]);

        if (value < key)
            low = mid + 1;
        else if (value > key)
            high = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

Probes. The `run` command at the start of a

The first statement in the function is used to setup the execution context: it provides values for the formal parameters of the function. Within the function, the `p` function accepts an expression as a first parameter. During execution the (successive) values of the expression are stored, indexed on the source position of the probe itself. The stored values are then put back into the source code as the second argument to `p`. Note that this

kind of probe is already more flexible than a simple view of runtime values of variables: `p` can be called with arbitrary expressions in the first argument, not just variables.

Discussion and Outlook

Implementing Live Literals

On the surface, live literals are a simple technique to improve the programming experience. Under the hood, however, some tricks are required. Here we discuss how live literals as presented in this document are implemented. At a very high-level, live literals require a dedicated interpreter for the parts of the code that contain the relevant literals. This interpreter is then invoked by the IDE at edit time to compute results and update the editor.

The first step involves identifying the context of the live literal in the code. For instance, in the spreadsheet example, the interpreter finds the `sheet` expression in the abstract syntax tree (AST) of the program. It then evaluates the expression, obtaining the result of the updated sheet. Note that the spreadsheet semantics is implemented as an ordinary library. The live literal interpreter solely acts as a broker between live literals and the source code in the editor. In our current implementation, the expression is evaluated using Javascript's `eval` function.

The source text in the editor is then patched to reflect the new result. The original AST of the live literal is compared with the new data value that it should represent. Whenever there is a change in a subtree, the corresponding source fragment is replaced with a textual representation of the new value. The fragment of text to be replaced is retrieved from accurate source locations on the AST. Whenever new literal data is inserted, a pretty printer ensures that the values are nicely represented in the source text.

Live literals benefit from additional origin tracking for the live literal expressions. For instance, the execution of the probe function `p` requires knowledge about where the function call occurs in the source code to accurately store intermediate results. In our Javascript implementation this information is retrieved from the stack trace contained in an exception that is thrown for this particular purpose. We expect that a more systematic approach to providing static source code origin information at run time will be a key enabler for additional live programming features.

Related Work

The more general concept of "live text" has been coined by Hancock in his PhD thesis[5] . Live literals can be seen as first level of live text where the interaction between programmer and programming environment is focused on data, as represented by programming language literals.

[5] Christopher Hancock, *Real-time programming and the big ideas of computational literacy*, MIT, 2003

Live literals are a step towards integrating aspects of the IDE with the source code of the program. The idea of bringing IDE and code closer is not a new idea. Case in point is the Smalltalk language, which really cannot be separated from the IDE. Smalltalk-like systems, however, represent the integration in the direction of the UI: the programmer mostly interacts with IDE,

and source code proper is limited to actual method bodies. In the other direction, there has been early work to explore the integration of command language and programming language[6]. Today, we tend not to make the distinction between command language and programming language anymore, but, ironically, live literals almost bring back the distinction by making the source code interactive, like a syntactically structured command line interface.

[6] Jan Heering and Paul Klint, Towards Monolingual Programming Environments, *ACM Transactions on Programming Languages and Systems*, Volume 7, Issue 2, pp. 183–213, 1985.

Full obliteration of the distinction between source code an GUI is represented by programming systems based on structure editing (als known as "projectional editing"), like MPS[7]. Such systems represent the opposite end of the spectrum regarding integration: there is no source code at all, the only interface is the graphical user interface. Although this provides arbitrary freedom to mix different notations, views and languages, the benefits of sharing and maintaining code using standard, textual tools are lost.

[7] Jetbrains, Meta Programming System (MPS)

Outlook: Source Code All the Things

Live literals are only the first step towards a more general concept of source code as an interactive medium. The scope of direct manipulation and immediate feedback is limited to literal expressions, but there are ample directions for generalizing the approach.

Two concrete directions for future work are *textual playgrounds* and *syntax for actions*. Textual playgrounds turn the source code itself into a live playground, scrapbook or REPL, where the programmer can try out snippets of code and immediately see the result. In fact, the concept of a read-eval-print-loop (REPL) could be directly supported within the source code. Syntax for actions captures the idea that meta-level operations on the program code or execution are invoked via programming constructs. A typical example is how breakpoints in Smalltalk are set using the `self halt.` statement. Another example would be to invoke refactorings using special language construct. An early experiment of the first author provided syntactic support for rename refactorings: appending the operator `->newName` to the name of an abstraction would perform the renaming. One can also imagine invoking jump-to-declaration, or show-references using a syntactic affordance.

The examples of live literals did not require any syntactic language extension of the host language. It is however interesting to ponder how programming languages could be designed from scratch to support live text. There is no inherent reason why a language could not have explicit syntactic support, for instance, to support textual playgrounds or syntax for actions. In a sense, this would create additional levels of interpretation in the source code that transcend, for instance, the standard distinction between comments (for humans) from code (for humans and the computer). A programming language's syntax would be stratified according to when certain constructs would be active, and for what purpose. Exploring the interaction between the strata is an interesting area of further research.

Conclusion

Live programming aims to improve the experience of programming with better techniques for interacting with programs and program execution. Live literals are a first step to increase feedback and direct manipulation, but without stepping

outside the realm of ordinary, textual source code. Live literals turn the otherwise mute and lame source code into an interactive medium for both input and output at development time. We have shown three examples illustrating live literals: live examples and tests, embedded spreadsheets, and probes. These mini case studies show the potential value of bringing liveness to the source code. Future work should focus on widening the scope of live literals to include other idioms of textual interactivity. Finally, live text poses challenges for language design in general, since it broadens the scope of traditional syntax and semantics, to include interaction as well.