Design and implementaiton of a live coding environment

Tomas Petricek
The Alan Turing Institute
London, United Kingdom
tomas@tomasp.net

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

1 Introduction

One of the aspects that make spreadsheet tools such as Excel more accessible than programming environments is their liveness. When you change a value in a cell in Excel, the whole spreadsheet is updated instantly and you see the new results immediately.

Increasing number of programming environments aim to provide the same live experience for more standard programming languages, but doing this is not easy. Fully recomputing the whole program after every single change is inefficient and calculating how a change in source code changes the result is extremely hard when the editor allows arbitrary manipulation of program text. For example, consider the following simple program that gets the release years of 10 most expensive movies in a data set movies:

```
let top = movies
  .sortBy(λx → x.getBudget()).take(10)
  .map(λx → x.getReleased().format("yyyy"))
```

A live coding environment shows a preview of the list of dates top and the programmer then changes the code by making 10 a variable and changing the date format to see the full date:

```
let count = 10
let top = movies
  .sortBy(λx → x.getBudget()).take(count)
  .map(λx → x.getReleased().format("dd-mm-yyyy"))
```

Ideally, the live coding environment should understand the change, reuse a cached result of the first two transformations (sorting and taking 10 elements) and only evaluate the last map and format the release dates of already computed top 10 movies.

This is not difficult to achieve if we represent the program in a structured way and allow the user to edit code only via well-understood primitive operations such as "extract variable" (which has no effect on the result) or "change constant value" (which forces recomputation of subsequent transformations). However, many programmers still prefer to edit programs as free form text. In this paper, we present the design and implementation of a live coding system that is capable of reusing previously evaluated expressions as in the example above, yet, is integrated into an ordinary text editor.

The motivation for this paper is twofold. First, implementing a live programming system requires a different way of thinking about compilers and interpreters than the one presented in classic programming language literature. Second, an increasing number of programming environments incorporate aspects of live programming [X,Y], yet relatively little has been written on their architecture and implementation.

PL'17, January 01–03, 2017, New York, NY, USA
2017.

```
let events =
  web.scrapeDatedLists("https://en.wikipedia.org/wiki/2016_in_science").explore
    .'filter data'.'Type is'.'Events[edit]'.then
    .'drop columns'.'drop Type'.then
    .'filter data'.'Entry contains'("Oxford").'or Entry contains'("Cambridge").then
    .'get the data'
```

web | filter by | drop columns | filter by | get the data

Date	Entry
12:00:00 AM, Nov 5, 2016	The University of Oxford AI Lab presents "LipNet", a new AI alg...
1:00:00 AM, Sep 29, 2016	A study led by the University of Cambridge finds that body-wor...
1:00:00 AM, Aug 4, 2016	A team at the University of Oxford achieves a quantum logic ga...
1:00:00 AM, Jun 13, 2016	Researchers at the University of Cambridge demonstrate a hyb...
12:00:00 AM, Mar 4, 2016	University of Cambridge scientists demonstrate that 'naïve' plu...

Figure 1. xxx

2 Live programming for data exploration

The Gamma introduction and what this tries to do. See pretty screenshot.

This is what we care about - it's not live programming for your typical haskell, but for R or Python.

3 Formalising live coding environment

In this section, we present a formalisation of a live coding environment for a small, expression-based programming language that supports let binding, member invocations and λ abstractions. This is the necessary minimum for data exploration as described in the previous section and it excludes constructs such as a mechanism for defining new objects.

$$e \quad = \quad \text{let } x = e \text{ in } e \mid \lambda x \rightarrow e \mid e.m(e, \ldots, e) \mid x \mid n$$

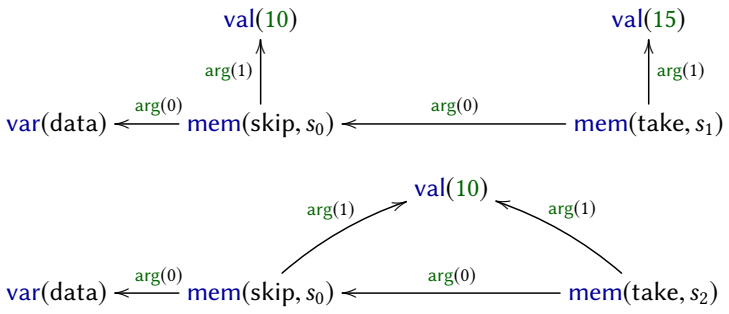
Here, m ranges over member names, x over variables and n over primitive values such as numbers. Function values can be passed as arguments to methods (provided by a type provider), but for the purpose of this paper, we do not need to be able to invoke them directly.

The problem with functions. In the context of live programming, let binding and member access are unproblematic. We can evaluate them and provide live preview for both of them, including all their sub-expressions. Function values are more problematic, because their sub-expressions cannot be evaluated. For example:

$$\text{let page} = \lambda x \rightarrow \text{movies.skip}(x * 10).\text{take}(10)$$

We can provide live preview for the movies sub-expression, but not for movies.skip($x * 10$) because we cannot obtain the value of x without running the rest of the program and analysing how the function is called later.

The method described in this paper does not provide live preview for sub-expressions that contain free variables (which are not global objects provided by a type provider), but we describe possible ways of doing so in Section X and more speculative design of live coding friendly fucntions in Section Y.



a.) The first graph is constructed from the following initial expression:

let x = 15 in
data.skip(10).take(x)

b.) The second diagram shows the updated graph after the programmer changes x to 10:

let x = 10 in
data.skip(10).take(x)

Figure 2. Dependency graphs formed by two steps of the live programming process.

3.1 Maintaining dependency graph

The key idea behind our implementation is to maintain a dependency graph with nodes representing individual operations of the computation that can be partially evaluated to obtain a preview. Each time the program text is modified, we parse it afresh (using an error-recovering parser) and bind the abstract syntax tree to the dependency graph.

We remember the previously created nodes of the graph. When binding a new expression to the graph, we reuse previously created nodes that have the same dependencies. For expressions that have a new structure, we create new nodes (using a fresh symbol to identify them).

The nodes of the graph serve as unique keys into a lookup table with previously evaluated operations of the computation. When a preview is requested, we use the node bound to the expression to find a preview, or evaluate it by first forcing the evaluation of all parents in the dependency graph.

Elements of the graph. The nodes of the graph represent individual operations to be computed. In our design, the nodes themselves are used as keys, so we attach a unique *symbol* to some of the nodes. That way, we can create two unique nodes representing, for example, access to a member named take which differ in their dependencies.

Furthermore, the graph edges are labelled with labels indicating the kind of dependency. For a method call, the labels are "first argument", "second argument" and so on. Formally:

$$
\begin{array}{rcll}
s & \in & \textit{Symbol} & \\
i & \in & \textit{Integer} & \\
n & \in & \textit{Primitive values} & \\
x & \in & \textit{Variable names} & \\
m & \in & \textit{Member names} & \\
\\
v & \in & \mathsf{val}(n) \mid \mathsf{var}(x) \mid \mathsf{mem}(m, s) \mid \mathsf{fun}(x, s) & (\textit{Vertices}) \\
l & \in & \mathsf{body} \mid \mathsf{arg}(i) & (\textit{Edge labels})
\end{array}
$$

The val node represents a primitive value and contains the value itself. Two occurences of 10 in the source code will be represented by the same node. Member access mem contains the member name, together with a unique symbol – two member access nodes with different dependencies will contain a different symbol. Dependencies of member access are labelled with arg indicating the index of the argument (the instance has index 0 and arguments start with 1).

Finally, nodes fun and var represent function values and variables bound by λ abstraction. For simplicity, we use variable names rather than de Bruijn indices and so renaming a bound variable foces recomputation.

Example graph. Figure 2 illustrates how we construct and update the dependency graph. Node representing take(x) depends on the argument – the number 15 – and the instance, which is a node representing skip(10). This, in turn, depends on the instance data and the number 10. Note that variables bound via let binding such as x do not appear as var nodes. The node using it depends directly on the node representing the result of the expression that is assigned to x.

After changing the value of x, we create a new graph. The dependencies of the node mem(skip, s_0) are unchanged and so the node is reused. This means that this part of the program is not recomputed. The arg(1) dependency of the take call changed and so we create a node mem(skip, s_2) with a new fresh symbol s_2. The preview for this node is then recomputed as needed using the already known values of its dependencies.

Reusing graph nodes. The binding process takes an expression and constructs a dependency graph, reusing existing nodes when possible. For this, we keep a lookup table of member access and function value nodes. The key is formed by a node kind (for disambiguation) together with a list of dependencies. A node kind is a member access or a function:

$$k \in \mathsf{fun}(x) \mid \mathsf{mem}(m) \qquad (\textit{Node kinds})$$

Given a lookup table Δ, we write $\Delta(k, [(n_1, l_1), \ldots, (v_n, l_n)])$ to perform a lookup for a node of a kind k that has dependencies $v_1, \ldots, v_n$ labelled with labels $l_1, \ldots, l_n$.

For example, when creating the graph in Figure 2 (b), we perform the following lookup for the skip member access:

$$\Delta(\mathsf{mem}(\mathsf{skip}), [(\mathsf{var}(\mathsf{data}), \mathsf{arg}(0)), (\mathsf{val}(10), \mathsf{arg}(1))])$$

$$\text{bind}_{\Gamma,\Delta}(e_0.m(e_1,\ldots,e_n)) = \quad (1)$$
$$v, (\{v\} \cup V_0 \cup \ldots \cup V_n, E \cup E_0 \cup \ldots \cup E_n)$$
$$\text{when } v_i, (V_i, E_i) = \text{bind}_{\Gamma,\Delta}(e_i)$$
$$\text{and } v = \Delta(\text{mem}(m), [(v_0, \text{arg}(0)), \ldots, (v_n, \text{arg}(n))])$$
$$\text{let } E = \{(v, v_0, \text{arg}(0)), \ldots, (v, v_n, \text{arg}(n))\}$$

$$\text{bind}_{\Gamma,\Delta}(e_0.m(e_1,\ldots,e_n)) = \quad (2)$$
$$v, (\{v\} \cup V_0 \cup \ldots \cup V_n, E \cup E_0 \cup \ldots \cup E_n)$$
$$\text{when } v_i, (V_i, E_i) = \text{bind}_{\Gamma,\Delta}(e_i)$$
$$\text{and } (\text{mem}(m), [(v_0, \text{arg}(0)), \ldots, (v_n, \text{arg}(n))]) \notin \text{dom}(\Delta)$$
$$\text{let } v = \text{mem}(m, s), s \text{ fresh}$$
$$\text{let } E = \{(v, v_0, \text{arg}(0)), \ldots, (v, v_n, \text{arg}(n))\}$$

$$\text{bind}_{\Gamma,\Delta}(\text{let } x = e_1 \text{ in } e_2) = v, (\{v\} \cup V \cup V_1, E \cup E_1) \quad (3)$$
$$\text{let } v_1, (V_1, E_1) = \text{bind}_{\Gamma,\Delta}(e_1)$$
$$\text{let } \Gamma_1 = \Gamma \cup \{(x, v_1)\}$$
$$\text{let } v, (V, E) = \text{bind}_{\Gamma_1,\Delta}(e_2)$$

$$\text{bind}_{\Gamma,\Delta}(n) = \text{val}(n), (\{\text{val}(n)\}, \emptyset) \quad (4)$$

$$\text{bind}_{\Gamma,\Delta}(x) = v, (\{v\}, \emptyset) \quad \text{when } v = \Gamma(x) \quad (6)$$

$$\text{bind}_{\Gamma,\Delta}(\lambda x \to e) = v, (\{v\} \cup V, \{e\} \cup E) \quad (7)$$
$$\text{when } \Gamma_1 = \Gamma \cup \{x, \text{var}(x)\}$$
$$\text{and } v_0, (V, E) = \text{bind}_{\Gamma_1,\Delta}(e)$$
$$\text{and } v = \Delta(\text{fun}(x), [(v_0, \text{body})])$$
$$\text{let } e = (v, v_0, \text{body})$$

$$\text{bind}_{\Gamma,\Delta}(\lambda x \to e) = v, (\{v\} \cup V, \{e\} \cup E) \quad (8)$$
$$\text{when } \Gamma_1 = \Gamma \cup \{x, \text{var}(x)\}$$
$$\text{and } v_0, (V, E) = \text{bind}_{\Gamma_1,\Delta}(e)$$
$$\text{and } (\text{fun}(x), [(v_0, \text{body})]) \notin \text{dom}(\Delta)$$
$$\text{let } v = \text{fun}(s, x), s \text{ fresh}$$
$$\text{let } e = (v, v_0, \text{body})$$

Figure 3. Rules of the binding process, whic constructs a dependency graph for an expression.

The lookup returns the node mem(skip, s_0) known from the previous step. We then perform the following lookup for the take member access:

$$\Delta(\text{mem}(\text{take}), [(\text{mem}(\text{skip}, s_0), \text{arg}(0)), (\text{val}(10), \text{arg}(1))])$$

In the previous graph, the argument of take was 15 rather than 10 and so this lookup fails. We then construct a new node mem(take, s_2) using a fresh symbol s_2.

3.2 Binding an expressions to a graph

When constructing the dependency graph, our implementation annotates the nodes of the abstract syntax tree with the nodes of the dependency graph, forming a mapping $e \to v$. For this reason, we call the process *binding*.

The process of binding is defined by the rules in Figure 3. The bind function is annotated with a lookup table Δ discussed in Section 3.1 and a variable context Γ. The variable context is a map from variable names to dependency graph nodes and is used for variables bound using let binding.

When applied on an expression e, binding $\text{bind}_{\Gamma,\Delta}(e)$ returns a dependency graph (V, E) paired with a node v corresponding to the expression e. In the graph, V is a set of nodes v and E is a set of labelled edges (v_1, v_2, l). We attach the label directly to the edge rather than keeping a separate colouring function as this makes the formalisation simpler.

Binding member access. In all rules, we recursively bind sub-expressions to get a dependency graph for each sub-expression and a graph node that represents it. The nodes representing sub-expressions are then used as dependencies for lookup into Δ, together with their labels. When binding a member access, we reuse an existing node if it is defined by Δ (1) or we create a new node containing a fresh symbol when the domain of Δ does not contain a key describing the current member access (2).

Binding let binding. For let binding (3), we first bind the expression e_1 assigned to the variable to obtain a graph node v_1. We then bind the body expression e_2, but using a variable context Γ_1 that maps the value of the variable to the graph node v_1. The variable context is used when binding a variable (6) and so all variables declared using let binding will be bound to a graph node representing the value assinged to the variable. The node bound to the overall let expression is then the graph node bound to the body expression.

Binding function values. If a function value uses its argument, we will not be able to evaluate its body. In this case, the graph node bound to a function will depend on a synthetic node var(x) that represents the variable with no value. When binding a function, we create the synthetic variable and add it to the variable context Γ_1 before binding the body. As with member access, the node representing a function may (7) or may not (8) be already present in the lookup table.

3.3 Edit and rebind loop

The binding process formalised in Section 3.2 specifies how to update the dependency graph after updated program text

$$\Delta_i(\text{mem}(m), [(v_0, \text{arg}(0)), \ldots, (v_n, \text{arg}(n))]) = v$$
$$\text{for all } \text{mem}(m, s) \in V$$
$$\text{such that } (\text{mem}(m, s), v_i, \text{arg}(i)) \in E \text{ for } i \in 0, .., n$$

$$\Delta_i(\text{fun}(x), [(v_1, \text{body})]) = v$$
$$\text{for all } \text{fun}(x, s) \in V$$
$$\text{such that } (\text{fun}(x, s), v_1, \text{body}) \in E$$

$$\Delta_i(v) = \Delta_{i-1}(v) \quad \text{(otherwise)}$$

Figure 4. Updating the node cache after binding a new graph

is parsed. During live coding, this is done repeatedly as the programmer edits code. Throughout the process, we maintain a series of lookup table states $\Delta_0, \Delta_1, \Delta_2, \ldots$. Initially, the lookup table is empty, i.e. $\Delta_0 = \emptyset$.

At a step i, we parse an expression e_i and calculate the new dependency graph and a node bound to the top-level expression using the previous Δ:

$$v, (V, E) = \mathsf{bind}_{\emptyset, \Delta_{i-1}}(e_i)$$

The new state of the node cache is then computed by adding newly created nodes from the graph (V, E) to the previous cache Δ_{i-1}. This is done for function and member nodes that contain unique symbols as defined in Figure 4. We do not need to cache nodes representing primitive values and variables as those do not contain symbols and will remain the same due to the way they are constructed.

4 Evaluating previews

5 Properties

References