Design and implementaiton of a live coding environment for data science

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

Abstract

Data science can be done by directly manipulating data using spreadsheets, or by writing data manipulation scripts using a programming language. The former is error-prone and does not scale, while the latter requires expert skills. Live coding has the potential to bridge this gap and make writing of transparent, reproducible scripts more accessible.

In this paper, we describe a live programming environment for data science that provides instant previews and contextual hints, while allowing the user to edit code in an unrestricted way in a text editor.

Using a text editor is challenging as any edit can significantly change the structure of code and fully recomputing previews after every change is too expensive. The presented interpreter partially evalautes programs as they are written and reuses results of previous evaluations, making it possible to provide instant feedback during development.

We formalise how programs are interpreted and how previews are computed, prove correctness of the previews and formally specify when can previews be reused. We hope this work will provide solid and easy to reuse foundations for the emerging trend of live programming environments.

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"))
```

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

A live coding environment shows a preview of the list of dates. Next, assume that the programmer modifies the code by making the constant 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 to differently format the release dates of already computed top 10 movies.

This is not difficult if we represent the program in a structured way and allow the user to edit code only via 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 prefer to edit programs as free form text.

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. Our main contributions are:

- We introduce The Gamma (Section 2), a simple live coding environment for data science. We review its design and implementation and explain how it bridges the gap between programming and spreadsheets.
- Implementing a live programming system requires different way of thinking about compilers and interpreters than the one presented in classic programming language literature. Our formalisation (Section 3) captures the essence of the new perspective.
- We formalise the evaluation of previews (Section 4) and prove that our evaluation and caching mechanism is produces correct previews (Section 5.2) and can effectively reuse partial results (Section 5.3).
- In more speculative conclusions (Section 6), we consider alternative language designs that would enable further live coding experiences, which are difficult to build using our current system.

We hope the architecture and its formal presentation in this paper can contribute important foundations to the growing and important trend of text-based live coding environments.

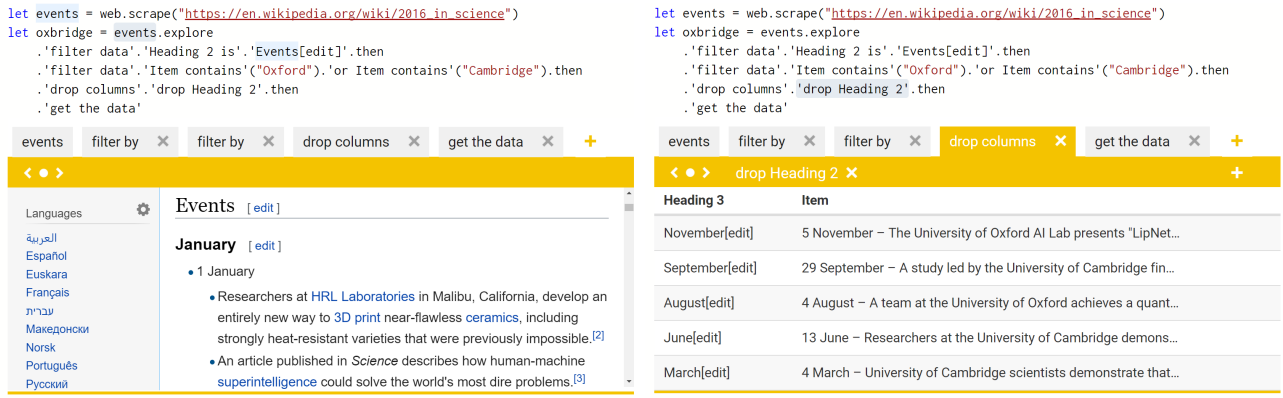


Figure 1. Scraping 2016 science events. Preview of the Wikipedia source page (left) and live filtering (right).

2 Live programming for data exploration

The Gamma aims to make basic data exploration accessible to non-experts, such as data journalists [X], who need to use data transparently. As such, we focus on simple scripts that can be written using tools such as Jupyter notebooks [X]. Those scripts follow a typical data science workflow [X]; they acquire and reformat data, run a number of analyses using the clean data and then create several visualizations.

An important property of data science workflow is that we work with readily available concrete data. Data scientists load inputs into memory and refer to it in subsequent REPL interactions. They might later wrap completed working code into a function (and run on multiple datasets), but do not start with functions. We reflect this pattern in our language.

In this section, we provide a brief overview of The Gamma, a simple live coding environment for data science. We discuss the language, type providers and the user interface, before focusing on the algorithms behind live previews in Section 3.

2.1 The Gamma scripting language

Figure 1 shows The Gamma script that scrapes items from a Wikipedia page, collects those marked as "Events" and filters them. The script illustrates two aspects of the scripting language used by The Gamma – its structure and its use of type providers for dot-driven data exploration.

The scripting language is not intended to be as expressive as, say, R or Python and so it has a very simple structure – scripts are a sequence of let-bindings (that obtain or transform data) or statements (that produce visualizations). This reflects the fact that we always work with concrete data and allows us to provide previews.

The most notable limitation of The Gamma is that the scripting language does not support top-level functions. This is not a problem for the simple scripts we consider, but it would be an issue for a general-purpose language. We discuss potential design for functions that would still be based on working with concrete data in Section 6.

2.2 Dot-driven data exploration

As ilustrated by the second let binding in Figure 1, many opertaions in The Gamma can be expressed using member access via dot. The underlying mechanism is based on type providers [X,Y] and the specific type provider used in this example has been described elsewhere [X].

Given a data source (scraped Wikipedia page), the type provider generates type with members that allow a range of transformations of the data such as grouping, sorting and filtering. Some of the members are based only on the schema (e.g. 'Item Contains' or 'Drop Heading 2'), but some may also be generated based on (a sample of) the dataset (e.g. the second member in 'Heading 2 is'.'Events[edit]').

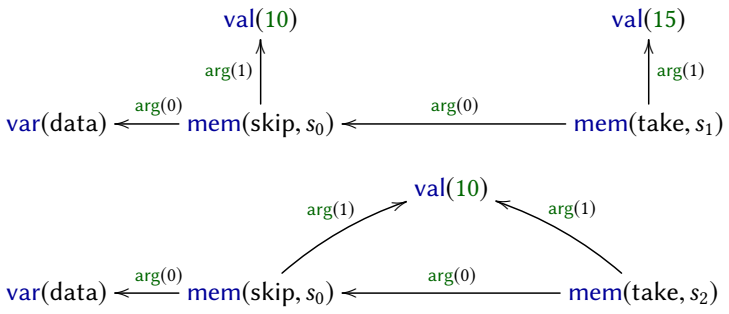
What matters for the purpose of this paper is the fact that most operations are expressed via member access matters. First, it means that we need to provide live previews for sub-expressions formed by a chain of member accesses. Second, it means that the result of any member access expression depends on the instance and, possibly, on the parameters provided for a method call.

2.3 Direct manipulation and live previews

The screenshot in Figure 1 show the editor as implemented in The Gamma. This includes live previews (discussed in this paper), but also an editor that provides spreadsheet-like interface for editing the data transformation script.

In our implementation, previews appears below the currently selected let binding. The preview can be of several types such as web page (left) or data table (right). In this paper, we describe how we evaluate scripts to obtain the resulting object and we ignore how such objects are rendered.

The Figure 1 also shows a special handling of expressions constructed using the pivot type provider. The editor recognises individual data transformations and provides a simple user interface for adding and removing transformations and changing their parameters that changes the source code accordingly. This aspect is not discussed in the present paper.



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 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.

It excludes constructs such as a mechanism for defining new objects as we assume that those are imported from the context through a mechanims such as type providers.

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

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.

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:

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

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).

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

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

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

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

$$\text{bind}_{\Gamma,\Delta}(x) = v, (\{v\}, \emptyset) \quad \text{when } v = \Gamma(x) \qquad (5)$$

$$\begin{aligned}
&\text{bind}_{\Gamma,\Delta}(\lambda x \rightarrow e) = v, (\{v\} \cup V, \{e\} \cup E) \qquad (6)\\
&\quad \text{when } \Gamma_1 = \Gamma \cup \{x, \text{var}(x)\}\\
&\quad \text{and } v_0, (V, E) = \text{bind}_{\Gamma_1,\Delta}(e)\\
&\quad \text{and } v = \Delta(\text{fun}(x), [(v_0, \text{body})])\\
&\quad \text{let } e = (v, v_0, \text{body})
\end{aligned}$$

$$\begin{aligned}
&\text{bind}_{\Gamma,\Delta}(\lambda x \rightarrow e) = v, (\{v\} \cup V, \{e\} \cup E) \qquad (7)\\
&\quad \text{when } \Gamma_1 = \Gamma \cup \{x, \text{var}(x)\}\\
&\quad \text{and } v_0, (V, E) = \text{bind}_{\Gamma_1,\Delta}(e)\\
&\quad \text{and } (\text{fun}(x), [(v_0, \text{body})]) \notin \text{dom}(\Delta)\\
&\quad \text{let } v = \text{fun}(s, x), s \text{ fresh}\\
&\quad \text{let } e = (v, v_0, \text{body})
\end{aligned}$$

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

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:

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

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:

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

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 $e \rightarrow v$. 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 $\mathsf{var}(x)$ 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 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.

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

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

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

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

4 Evaluating previews

The mechanism for constructing dependency graphs defined in Section 3 makes it possible to provide live previews when editing code without recomputing the whole program each time the source code changes.

The nodes in the dependency graph correspond to individual operations that will be performed when running the program. When the dependencies of an operation do not change while editing code, the subsequent dependency graph will reuse a node used to represent the operation.

Our live editor keeps a map from graph nodes to live previews, so a new preview only needs to be computed when a new node appears in the dependency graph (and the user moves the cursor to a code location that corresponds to the node). This section describes how previews are evaluated.

Previews and delayed previews. As discussed in Section 3, the body of a function cannot be easily evaluated to a value if it uses the bound variable. We do not attempt to "guess" possible arguments and, instead, provide a full preview only for sub-expressions with free variables bound by a let binding. For a function body that uses the bound variable, we obtain a *delayed preview*, which is an expression annotated with a list of variables that need to be provided before the expression can be evaluated. We use the following notation:

$$p \in n \mid \lambda x \to e \qquad \text{(\textit{Fully evaluated previews})}$$
$$d \in p \mid [\![e]\!]_\Gamma \qquad \text{(\textit{Evaluated and delayed previews})}$$

Here, p ranges over fully evaluated values. It can be either a primitive value (such as number, string or an object) or a function value with no free variables. A possibly delayed preview d can then be either evaluated preview p or an expression e that requires variables Γ. For simplicity, we use an untyped language and so Γ is a list of variables $x_1, \ldots, x_n$.

Evaluation and splicing. In this paper, we omit the specifics of the underlying programming langauge and we focus on the live coding mechanism. However, we assume that the language is equipped with an evaluation reduction $e \rightsquigarrow p$ that reduces a closed expression e into a value p.

For delayed previews, we construct a delayed expression using splicing. For example, assuming we have a delayed previews $[\![e_0]\!]_x$ and $[\![e_1]\!]_y$. If we need to invoke a member m on e_0 using e_1 as an argument, we construct a new delayed preview $[\![e_0.m(e_1)]\!]_{x,y}$. This operation is akin to expression splicing from meta-programming [X,Y] and can be more formally captured by Contextual Modal Type Theory (CMTT) as outlined below.

Evaluation of previews. The evaluation of previews is defined in Figure 5. Given a dependency graph (V, E), we define a relation $v \Downarrow d$ that evaluates a sub-expression corresponding to the node v to a (possibly delayed) preview d.

The auxiliary relation $v \Downarrow_{\mathsf{lift}} d$ always evaluates to a delayed preview. If the ordinary evaluation returns a delayed

$$\text{(lift-expr)}\ \frac{v \Downarrow [\![e]\!]_\Gamma}{v \Downarrow_{\text{lift}} [\![e]\!]_\Gamma} \qquad \text{(fun-val)}\ \frac{(\text{fun}(x,s), v, \text{body}) \in E \quad v \Downarrow p}{\text{fun}(x,s) \Downarrow \lambda x \to p}$$

$$\text{(lift-prev)}\ \frac{v \Downarrow p}{v \Downarrow_{\text{lift}} [\![p]\!]_\emptyset} \qquad \text{(fun-bind)}\ \frac{(\text{fun}(x,s), v, \text{body}) \in E \quad v \Downarrow [\![e]\!]_x}{\text{fun}(x,s) \Downarrow \lambda x \to e}$$

$$\text{(val)}\ \frac{}{\text{val}(n) \Downarrow n} \qquad \text{(fun-expr)}\ \frac{(\text{fun}(x,s), v, \text{body}) \in E \quad v \Downarrow [\![e]\!]_{x,\Gamma}}{\text{fun}(x,s) \Downarrow [\![\lambda x \to e]\!]_\Gamma}$$

$$\text{(var)}\ \frac{}{\text{var}(x) \Downarrow [\![x]\!]_x}$$

$$\text{(mem-val)}\ \frac{\forall i \in \{0 \ldots k\}.(\text{mem}(m,s), v_i, \text{arg}(i)) \in E \quad v_i \Downarrow p_i \quad p_0.m(p_1, \ldots, p_k) \rightsquigarrow p}{\text{mem}(m,s) \Downarrow p}$$

$$\text{(mem-expr)}\ \frac{\forall i \in \{0 \ldots k\}.(\text{mem}(m,s), v_i, \text{arg}(i)) \in E \quad \exists j \in \{0 \ldots k\}.v_j \not\Downarrow p_j \quad v_i \Downarrow_{\text{lift}} [\![e_i]\!]_{\Gamma_i}}{\text{mem}(m,s) \Downarrow [\![e_0.m(e_1, \ldots, e_k)]\!]_{\Gamma_0,\ldots,\Gamma k}}$$

Figure 5. Rules that define evaluation of previews over a dependency graph for an expression

preview, so does the auxiliary relation (*lift-expr*). If the ordinary evaluation returns a value, the value is wrapped into a delayed preview requiring no variables (*lift-prev*).

Graph node representing a value is evaluated to a value (*val*) and a graph node representing an unbound variable is reduced to a delayed preview that requires the variable and returns its value (*var*).

For member access, we distinguish two cases. If all arguments evaluate to values (*member-val*), then we use the evaluation relation $\rightsquigarrow$, immediately evaluate the member access and produce a value. If one of the arguments is delayed (*member-expr*), because the member access is in the body of a lambda function, then we produce a delayed member access experession that requires the union of the variables required by the individual arguments.

Finally, the evaluation of function values is similar, but requires three cases. If the body can be reduced to a value with no unbound variables (*fun-val*), we return a lambda function that returns the value. If the body requires only the bound variable (*fun-bind*), we return a lambda function with the delayed preview as the body. If the body requires further variables, the result is a delayed preview containing a function.

Caching previews. For simplicity, the relation $\Downarrow$ in Figure 5 does not specify how previews are cached and linked to graph nodes. In practice, this is done by maintaining a lookup table $v \to p$ from graph nodes v to (possibly delayed) previews p. Whenever the $\Downarrow$ relation is used to obtain a preview for a graph node, we first attempt to find an already evaluated preview using the lookup table. If the preview has not been previously evaluated, we evaluate it and add it to the lookup table.

This means that evaluated previews can be reused in two ways. First, multiple nodes can depend on one sub-graph in a single dependency graph (e.g. if the same sub-expression appears twice in the program). Second, the keys of the lookup table are graph nodes and nodes are reused when a new dependency graph is constructed after the user edits the source code.

Semantics of delayed previews. The focus of this paper is on the design and implementation of a live coding environment, but it is worth noting that the structure of delayed previews is closely linked to the work on Contextual Modal Type Theory (CMTT) [X] and comonads [Y].

In CMTT, $[\Psi]A$ denotes that a proposition A is valid in context Ψ, which is closely related to our delayed preivews written as $[\![A]\!]_\Psi$. CMTT defines rules for composing context-dependent propositions that would allow us to express the splicing operation used in (*mem-expr*). In categorical terms, the context-dependent proposition can be modelled as an indexed comonad [X]. The evaluation of a preview with no context dependencies (built implicitly into our evaluation rules) corresponds to the counit operation of a comonad and would be explicitly written as $[\![A]\!]_\emptyset \to A$.

5 Properties of preview evaluation

The dependency graph makes it possible to cache partial results when evaluating previews. The construction of dependency graphs and evaluation of previews needs to have two main properties. First, if we evaluate a preview using dependency graph with caching, it should be the same as the value we would obtain by evaluating the expression directly. Second, the evaluation of previews using dependency graphs should – in some cases – reuse previously evaluated partial results. In other words, this section shows that our mechanism is both correct and implements a useful optimization.

5.1 Host language

The method in this paper does not rely on the particulars of how the underlying programming language works. For this reason, we do not give a concrete definition of how member access $e.m(e_1, \ldots, e_n)$ reduces to a value.

We require that the host language allows eliminating let bindings without changing the meaning of the program. That is

[Rightsquigarrow allows beta anywhere? then it works...]

Definition 1 (Host language reduction)**.** The reduction $\rightsquigarrow$ reduces host language expression such that:

- Given an expression e and e' such that e' is obtained from e by replacing any sub-expression let $x = e_1$ in e_2 with the expression $e_2[x \leftarrow e_1]$, written as $e \rightsquigarrow_{\text{let}} e'$, then if $e \rightsquigarrow e''$ then also $e' \rightsquigarrow e''$.
- Compositionality, i.e. if $e \rightsquigarrow e'$ and $C[e] \rightsquigarrow e''$ then also $C[e'] \rightsquigarrow e''$.
- Extensionality – if $C1[e1] \rightsquigarrow e$ and $C1[e2] \rightsquigarrow e$ then $C2[f\,ne1] \rightsquigarrow ee$ and $C2[f\,ne2] \rightsquigarrow ee$

We don't care about non-termination here. (so, does $\rightsquigarrow$ do small step or big step reduction?)

5.2 Correctness of previews

First is corrctness (Theorem **??**) – no matter how a graph is constructed, when we use it to evaluate a preview for an expression, the preview is the same as the value we would obtain by evaluating the expression directly. Second is determinacy (Theorem 2) – if we cache a preview for a graph node and update the graph, the preview would still be the same when evaluated using an updated graph.

This is the correctness and determinacy property discussed in Section 5.2.

Eliminating let bindings does not change the result in the host language. It also yields the same dependency graph.

Lemma 1 (Let elimintion)**.** *Given an expression e_1 such that $e_1 \rightsquigarrow_{\text{let}} e_2$ and a lookup table Δ then if $v_1, (V_1, E_1) = \text{bind}_{\emptyset,\Delta}(e_1)$ and $v_2, (V_2, E_2) = \text{bind}_{\emptyset,\Delta}(e_2)$ then it holds that $v_1 = v_2$ and also $(V_1, E_1) = (V_2, E_2)$.*

This simplifies the correctness proof as we only need to consider expressions not containing let bindings (if there were let bindings, both the result and the preview would be equivalent to one that would be obtained if the let bindings were elimninated)

Now, if we have any let-free expression, we show that the preview obtained via the dependency graph is a value that is the same as the one we'd get by directly evaluating the expression.

xxx

Theorem 2 (Determinacy)**.** *Downarrow will always evaluate the same, if we add things to a graph in the defined way. (this means that we can cache)*

[Only dependency graph matters, not the history and symbols] [Symbols are effectively hashes of dependencies]

Lemma X - lookup with red mem(m) returns blue mem(m, s)

Theorem 3 (Let-free correctness)**.** *Given an expression e that has no free variables and does not contain let bindings, together with a lookup table Δ obtained from any sequence of expressions according to Figure 4 let $v, (V, E) = \text{bind}_{\emptyset,\Delta}(e)$. If $v \Downarrow d$ over a graph (V, E) then $d = p$ for some p and $e \rightsquigarrow p$.*

Proof. First note that, when combining recursively constructed sub-graphs, the bind operation adds new nodes and edges leading from those new nodes. Therefore, an evaluation using $\Downarrow$ over a sub-graph will also be valid over the new graph.

Next, we prove a more general property using induction showing that for e such that $v, (V, E) = \text{bind}_{\emptyset,\Delta}(e)$:

a. If $FV(e) = \emptyset$ then $v \Downarrow p$ for some p and $e \rightsquigarrow p$
b. Otherwise, $v \Downarrow [\![e_p]\!]_{FV(e)}$ for some e_p and for any evaluation context $C[-]$ such that $FV(C[e_p]) = \emptyset$ it holds that if $C[e_p] \rightsquigarrow v$ for some v then $C[e] \rightsquigarrow v$.

The proof is done by induction over the binding process, which follows the structure of the expression e and can be found in Appendix A. □

Theorem 4 (Correctness)**.** *(Same as let-free version, but using let elimination to say that this works for any expression)*

5.3 Reuse of previews

* Inlining let leads to an equivalent dependency graph

* Given two expressions, if we inline lets and they are structurally different, then their dependency graphs are structurally different

Something about how dependency graph bits are reused correctly, i.e. if there is a source code change that can change the result, the graph will change.

Something about various code edit operations that will not cause recomputation.

6 Takeaways

7 Related and future

Integrate with Python/R via Jupyter

References

A Appendix

Theorem 5 (Let-free correctness)**.** *Given an expression e that has no free variables and does not contain let bindings, together with a lookup table Δ obtained from any sequence of expressions according to Figure 4 let $v, (V, E) = \mathsf{bind}_{\emptyset,\Delta}(e)$. If $v \Downarrow d$ over a graph (V, E) then $d = p$ for some p and $e \rightsquigarrow p$.*

Proof. First note that, when combining recursively constructed sub-graphs, the bind operation adds new nodes and edges leading from those new nodes. Therefore, an evaluation using $\Downarrow$ over a sub-graph will also be valid over the new graph.

Next, we prove a more general property using induction showing that for e such that $v, (V, E) = \mathsf{bind}_{\emptyset,\Delta}(e)$:

a. If $FV(e) = \emptyset$ then $v \Downarrow p$ for some p and $e \rightsquigarrow p$
b. Otherwise, $v \Downarrow [\![e_p]\!]_{FV(e)}$ for some e_p and for any evaluation context $C[-]$ such that $FV(C[e_p]) = \emptyset$ it holds that if $C[e_p] \rightsquigarrow v$ for some v then $C[e] \rightsquigarrow v$.

The proof is done by induction over the binding process, which follows the structure of the expression e:

(1) $\mathsf{bind}_{\Gamma,\Delta}(e_0.m(e_1, \ldots, e_n))$ – Here $e = e_0.m(e_1, \ldots, e_n)$, v_i are graph nodes obtained by induction for expressions e_i and $\{(v, v_0, \mathsf{arg}(0)), \ldots, (v, v_n, \mathsf{arg}(n))\} \subseteq E$ From Lemma 5.2, $v = \mathsf{mem}(m, s)$ for some s.

If $FV(e) = \emptyset$, then $v_i \Downarrow p_i$ for $i \in 0 \ldots n$ and $v \Downarrow p$ using (*mem-val*) such that $p_0.m(p_1, \ldots, p_n) \rightsquigarrow p$. From induction hypothesis, $e_i \rightsquigarrow p_i$ and so, using compositionality of $\rightsquigarrow$, $e_0.m(e_1, \ldots, e_n) \rightsquigarrow p$.

If $FV(e) \neq \emptyset$, then $v_i \Downarrow_{\mathsf{lift}} [\![e'_i]\!]$ for $i \in 0 \ldots n$ and $v \Downarrow [\![e'_0.m(e'_1, \ldots, e'_n)]\!]_{FV(e)}$ using (*mem-expr*). From induction hypothesis, for any $C[-]$, if $C[e'_i] \rightsquigarrow v_i$ then $C[e_i] \rightsquigarrow v_i$. Using compositionality, it also holds that for any $C[-]$, if $C[e'_0.m(e'_1, \ldots, e'_n)] \rightsquigarrow v$ then also $C[e_0.m(e_1, \ldots, e_n)] \rightsquigarrow v$.

(2) $\mathsf{bind}_{\Gamma,\Delta}(e_0.m(e_1, \ldots, e_n))$ – This case is similar to (1), except that the fact that $v = \mathsf{mem}(m, s)$ holds by construction, rather than using Lemma 5.2.

(3) We assume that the expression e does not include let bindings and so this case never happens.

(4) $\mathsf{bind}_{\Gamma,\Delta}(n)$ – In this case $e = n$ and $v = \mathsf{val}(n)$. The preview of $\mathsf{val}(n)$ is evaluated to n using the (*val*) case.

(5) $\mathsf{bind}_{\Gamma,\Delta}(x)$ – The initial Γ is empty and there are no let bindings, so x must have been added to Γ by case (6) or (7). Hence, $v = \mathsf{var}(x)$. Using (*var*) $v \Downarrow [\![x]\!]_x$ and so $e_p = e = x$ and the second case (b.) trivially holds.

(6) $\mathsf{bind}_{\Gamma,\Delta}(\lambda x \to e)$ – Assume v_b is a graph node representing the body. The evaluation can use one of three rules:

If $FV(e) = \emptyset$ then $v_b \Downarrow p_b$ for some p_b and $v \Downarrow \lambda x.p_b$ using (*fun-val*). From induction $e_b \rightsquigarrow p_b$ and so, using extensionality of $\rightsquigarrow$ also $\lambda x.e_b \rightsquigarrow \lambda x.p_b$.

If $FV(e) = \{x\}$ then $v_b \Downarrow [\![e_b]\!]_x$ for some e_b and $v \Downarrow \lambda x.e_b$ using (*fun-bind*). From induction, for any $C[-]$, if $C[e_b] \rightsquigarrow v$ then also $C[e] \rightsquigarrow v$. By extensionality, using an empty context $C[-] = -$ it holds that if $\lambda x.e \rightsquigarrow \lambda x.e_b$.

Otherwise, $v_b \Downarrow [\![e_b]\!]_{x,\Gamma}$ for some e_b and $v \Downarrow [\![\lambda x.e_b]\!]_\Gamma$ using (*fun-expr*). From induction, for any $C[-]$, if $C[e_b] \rightsquigarrow v$ then also $C[e] \rightsquigarrow v$. By extensionality, if $C'[\lambda x.e_b] \rightsquigarrow v$ then also $C'[\lambda x.e] \rightsquigarrow v$. □