Design and implementation of a live coding environment for data science

Tomas Petricek

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.

Supporting 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. We present a technique that allows the type checker and the interpreter to run on the fly, while code is written, and reuse results of previous runs. This makes it possible to efficiently provide instant feedback and live previews 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 believe this work provides 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 [26], 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. 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 [20, 31] and allow the user to edit code via known 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 6.2) and can effectively reuse partial results (Section 6.3).
- We follow the same method to implement a type checker (Section 5) for our language that is capable of reusing previous results. This makes it possible to efficiently support asynchronously provided types (Section 5.1).
- In more speculative conclusions (Section 7), 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 crucial foundations for the growing and important trend of text-based live coding environments.

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

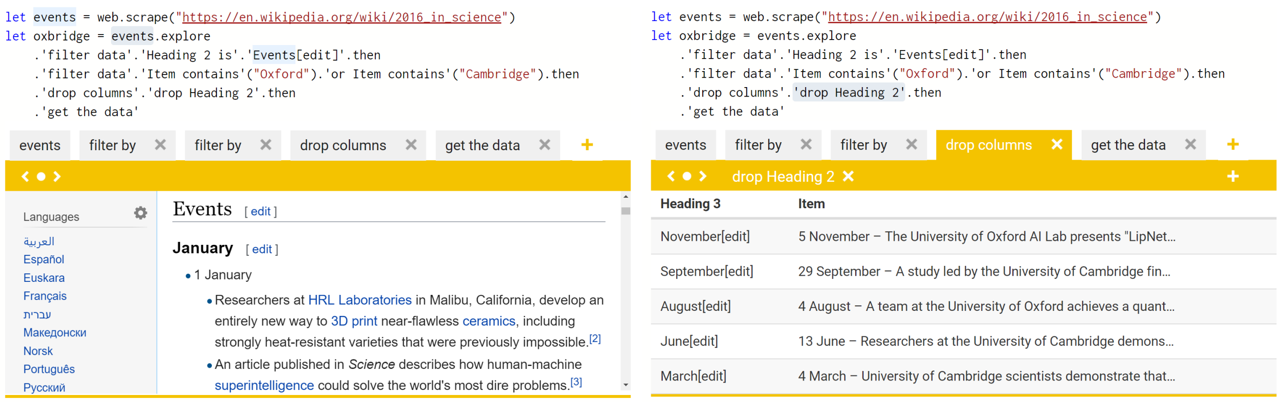


Figure 1. Scraping 2016 science events. Preview of the Wikipedia source page (left) and live filtering (right).

2 Live programming for data exploration

The Gamma aims to make basic data exploration accessible to non-experts, such as data journalists [12], who need to use data transparently. As such, we focus on simple scripts that can be written using tools such as Jupyter notebooks [15]. Those scripts follow a typical data science workflow [13]; 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 7.

2.2 Dot-driven data exploration

As illustrated by the second let binding in Figure 1, many operations in The Gamma can be expressed using member access via dot. The underlying mechanism is based on type providers [3, 29]. The specific type provider used in the above example has been described elsewhere [23].

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]').

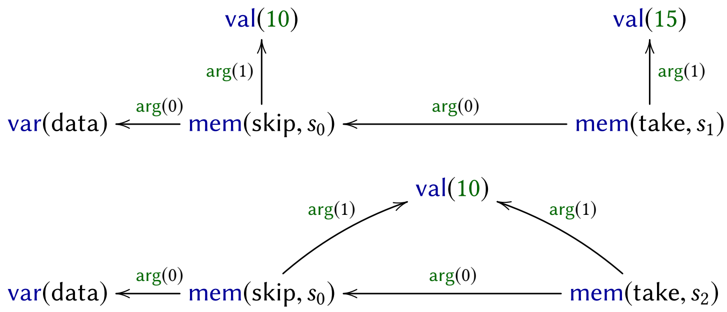
What matters for the purpose of this paper is the fact that most operations are expressed via member access matters. First, it means that we need to provide live previews for sub-expressions formed by a chain of member accesses. Second, it means that the result of any member access expression depends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

In this section, we present a formalisation of a live coding infrastructure 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 mechanism such as type providers.

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

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:

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

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 obtains delayed previews for sub-expressions that contain free variables (which could be fully evaluated if values for free variables were provided), but we describe a more speculative design of live coding friendly functions in Section 7.

3.1 Maintaining dependency graph

The key idea behind our implementation is to maintain a dependency graph [16] 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 labeled with labels indicating the kind of dependency. For a method call, the labels are "first argument", "second argument" and so on. Formally:

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

The val node represents a primitive value and contains the value itself. Two occurrences 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 labeled with arg indicating the index of the argument (the instance has index 0 and arguments start with 1).

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

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

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

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

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

$$\begin{array}{l}\mathsf{bind}_{\Gamma,\Delta}(\lambda x \to e) = v, (\{v\} \cup V, \{e\} \cup E) \\ \quad \text{when } \Gamma_1 = \Gamma \cup \{x, \mathsf{var}(x)\} \\ \quad \text{and } v_0, (V, E) = \mathsf{bind}_{\Gamma_1,\Delta}(e) \\ \quad \text{and } v = \Delta(\mathsf{fun}(x), [(v_0, \mathsf{body})]) \\ \quad \text{let } e = (v, v_0, \mathsf{body})\end{array} \tag{6}$$

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

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 forces 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 $\mathsf{mem}(\mathsf{skip}, s_0)$ are unchanged and so the node is reused. This means that this part of the program is not recomputed. The $\mathsf{arg}(1)$ dependency of the take call changed and so we create a node $\mathsf{mem}(\mathsf{skip}, s_2)$ 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:

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

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 $v_1, \ldots, v_n$ labeled 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:

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

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

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

In the previous graph, the argument of take was 15 rather than 10 and so this lookup fails. We then construct a new node $\mathsf{mem}(\mathsf{take}, s_2)$ 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 $\mathsf{bind}_{\Gamma,\Delta}(e)$ 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 labeled 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 assigned 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.

$\mathsf{update}_{V,E}(\Delta_{i-1}) = \Delta_i$ such that:

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

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

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

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:

$$\begin{array}{lcll}
p & \in & n \mid \lambda x \to e & (\textit{Fully evaluated previews})\\
d & \in & p \mid [\![e]\!]_\Gamma & (\textit{Evaluated and delayed previews})
\end{array}$$

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 language 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 [28, 32] and can be more formally captured by Contextual Modal Type Theory (CMTT) as outlined below.

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

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

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

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

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

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

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

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 $v \Downarrow_{\text{lift}} d$ always evaluates to a delayed preview. If the ordinary evaluation returns a delayed 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 expression that requires the union of the variables required by the individual arguments.

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.

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 from graph nodes v to (possibly delayed) previews p. Whenever $\Downarrow$ 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.

The evaluated previews can be reused in two ways. First, multiple nodes can depend on one sub-graph in a single dependency graph (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) [19] and comonads [9].

In CMTT, $[\Psi]A$ denotes that a proposition A is valid in context Ψ, which is closely related to our delayed previews 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 modeled as an indexed comonad [10, 18]. 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 Type checking

Evaluating live previews can be an expensive operations, so being able to cache partial previews is a must for a live coding environment. Type checking is typically fast, so the main focus of this paper is on live previews. However, asynchronous type providers in The Gamma (Section 5.1) can make type

$$\begin{aligned}
&\mathsf{bind}_{\Gamma,\Delta,c}(e_0.m(e_1,\ldots,e_n)) = \qquad (2b)\\
&\quad v, (\{v\} \cup V_0 \cup \ldots \cup V_n, E \cup E_0 \cup \ldots \cup E_n)\\
&\text{when } v_0, (V_0,E_0) = \mathsf{bind}_{\Gamma,\Delta,\perp}(e_0)\\
&\text{and } c_i = (v_0, \mathsf{callsite}(m,i)) \qquad (i \in 1\ldots n)\\
&\text{and } v_i, (V_i,E_i) = \mathsf{bind}_{\Gamma,\Delta,c_i}(e_i) \qquad (i \in 1\ldots n)\\
&\text{and } (\mathsf{mem}(m), [(v_0, \mathsf{arg}(0)),\ldots,(v_n,\mathsf{arg}(n))]) \notin \mathrm{dom}(\Delta)\\
&\text{let } v = \mathsf{mem}(m,s), s \text{ fresh}\\
&\text{let } E = \{(v, v_0, \mathsf{arg}(0)),\ldots,(v, v_n, \mathsf{arg}(n))\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{bind}_{\Gamma,\Delta,(v_c,l_c)}(\lambda x \to e) = v, (\{v\} \cup V_0, E \cup E_0) \qquad (7b)\\
&\text{when } (\mathsf{var}(x), [(v_c, l_c)]) \notin \mathrm{dom}(\Delta)\\
&\text{let } v_x = \mathsf{var}(x, s_x), s_x \text{ fresh}\\
&\text{and } \Gamma_1 = \Gamma \cup \{x, v_x\}\\
&\text{and } v_0, (V_0,E_0) = \mathsf{bind}_{\Gamma_1,\Delta,\perp}(e)\\
&\text{when } (\mathsf{fun}(x), [(v_0, \mathsf{body}), (v_c, l_c)]) \notin \mathrm{dom}(\Delta)\\
&\text{let } v = \mathsf{fun}(x, s_f), s_f \text{ fresh}\\
&\text{let } E = \{(v, v_0, \mathsf{body}), (v, v_c, l_c), (v_x, v_c, l_c)\}
\end{aligned}$$

Figure 6. Revised binding rules, tracking call sites of function values.

checking time consuming, and so we use the dependency graph also for type checking (Section 5.3). Type checking lambda functions (Section 5.2) requires a slight extension of the model discussed in Section 3.

5.1 Asynchronously provided types

Data available in The Gamma can be defined using several kinds of type providers. The type provider used in Figure 1 as asynchronous [30]. It downloads the sample URL and generates types based on the contents of the web page. The parameter to web.scrape is a static parameter and is evaluated during type-checking. We omit details in this paper, but we note this works similarly to F# type providers [29].

Type providers can also be implemented as REST services [7] to allow anyone implement a data source in the language of their choice. In this case, each member of a call-chain returns a type that is generated based on the result of an HTTP request. For example, when the user types worldbank (to access information about countries), the type provider makes a request to http://thegamma-services.azurewebsites.net/worldbank, which returns two members:

```
[ {"name": "byYear", "returns":
    {"kind": "nested", "endpoint": "/pickYear"}}
  {"name": "byCountry", "returns":
    {"kind": "nested", "endpoint": "/pickCountry"}} ]
```

This indicates that worldbank has members byYear and byCountry. If the user types worldbank.byCountry, a request is made to the specified URL http://thegamma-services.azurewebsites.net/worldbank/pickCountry:

```
[ {"name": "Andorra", "trace": [""country=AR""],
   "returns": {"kind": "nested", "endpoint": "/pickTopic"}}
  {"name": "Afghanistan", "trace": [""country=AF""],
   "returns": {"kind": "nested", "endpoint": "/pickTopic"}}, ..]
```

This returns a list of countries which can then be accessed as members via worldbank.byCountry.Andorra, etc.

This is one reason for why type checking in The Gamma can be time consuming. Other type providers may perform other more computationally intensive work to provide types and so it is desirable to reuse type-checking results during live coding. The rest of this section shows how this is done using the dependency graph discussed in Section 3.

5.2 Revised binding of functions

The Gamma script supports lambda functions, but only in a limited way. A function can be passed as a parameter to a method, which makes type checking of functions easier. For example, consider:

$$\mathsf{movies.sortBy}(\lambda x \to x.\mathsf{getBudget}())$$

If movies is a collection of Movie objects, the type of the lambda function must be Movie → bool and so the type of x is Movie. This is similar to type checking of lambda functions in C# [33], where type is also inferred from the context (or has to be specified explicitly).

We do not currently allow lambda functions as stand-alone let-bound values. This could be done by requiring explicit types, or introducing polymorphism, but it was not necessary for the limited domain of non-expert data exploration.

Dependency graph for functions. In the binding process specified in Section 3, a variable is a leaf of the dependency graph. In the revised model, it depends on the context in which it appears. A new edge labeled $\mathsf{callsite}(m,i)$ indicates that the source node is the input variable of a function passed as the i^{th} argument to the m member of the expression represented by the target node. A node representing function is linked to the call site using the same edge.

Figure 7 shows the result of binding $o.m(\lambda x \to x)$. Both $\mathsf{fun}(x, s_2)$ and $\mathsf{var}(x, s_1)$ now depend on the node representing o. The new callsite edge makes it possible to type-check function and variable nodes just using their dependencies. As before, the member invocation $\mathsf{mem}(m, s_0)$ depends on the instance using $\mathsf{arg}(0)$ and on its argument using $\mathsf{arg}(1)$.

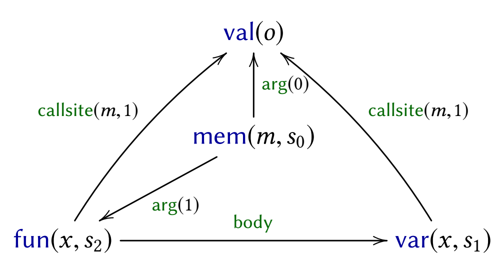


Figure 7. Dependency graph for $o.m(\lambda x \to x)$

$$\text{(val)}\ \frac{\Sigma(n) = \alpha}{\mathsf{val}(n) \vdash \alpha} \qquad \text{(var)}\ \frac{(\mathsf{var}(x,s), v, \mathsf{callsite}(m,i)) \in E \quad v \vdash (..,m : (\tau_1,\ldots,\tau_k) \to \tau,..) \quad \tau_i = \tau' \to \tau''}{\mathsf{var}(x,s) \vdash \tau'}$$

$$\text{(fun)}\ \frac{\{(\mathsf{fun}(x,s), v_b, \mathsf{body}), (\mathsf{var}(x,s), v_c, \mathsf{callsite}(m,i))\} \subseteq E \quad v_c \vdash (..,m : (\tau_1,\ldots,\tau_k) \to \tau,..) \quad \tau_i = \tau' \to \tau'' \quad v_b \vdash \tau''}{\mathsf{fun}(x,s) \vdash \tau' \to \tau''}$$

$$\text{(mem)}\ \frac{\forall i \in \{0\ldots k\}.(\mathsf{mem}(m,s), v_i, \mathsf{arg}(i)) \in E \quad v_0 \vdash (..,m : (\tau_1,\ldots,\tau_k) \to \tau,..) \quad v_i \vdash \tau_i}{\mathsf{mem}(m,s) \vdash \tau}$$

Figure 8. Rules that define evaluation of previews over a dependency graph for an expression

Revised binding process. For the revised binding process, we introduce a new edge label callsite. Variable nodes now have dependencies and so we cache them and attach a symbol s to var, so we also introduce a node kind var as part of a lookup key for Δ.

The bind function now has a parameter c, in addition to Γ and Δ, which represents the context in which the binding happens. This is either a member invocation (labeled with instance node v and callsite label l), or not a member invocation written as $\perp$. The updated definitions are:

$$\begin{array}{lcll} v & \in & \mathsf{val}(n) \mid \mathsf{var}(x,s) \mid \mathsf{mem}(m,s) \mid \mathsf{fun}(x,s) & (\textit{Vertices}) \\ l & \in & \mathsf{body} \mid \mathsf{arg}(i) \mid \mathsf{callsite}(m,i) & (\textit{Edge labels}) \\ k & \in & \mathsf{fun}(x) \mid \mathsf{mem}(m) \mid \mathsf{var}(x) & (\textit{Node kinds}) \\ c & \in & \perp \mid (v,l) & (\textit{Call sites}) \end{array}$$

The key parts of the revised definition of the bind function are shown in Figure 6. We now write $\mathsf{bind}_{\Gamma,\Delta,c}$ where c represents the context in which the binding occurs. This is set to $\perp$ in all cases, except when binding arguments of a member call. In (2b), we first recursively bind the instance node using $\perp$ as the context and then bind all the arguments using $(v_0, \mathsf{callsite}(m,i))$ as the context for i^{th} argument. The rest is as in case (2) before. The case (1) is updated similarly and is not shown here for brevity.

When binding a function (7b), we now also store variable nodes in Δ and so we check if a variable with the given call site exists. If no, we create a fresh node $\mathsf{var}(x, s_x)$. The node is added to Γ as before. At the end, we now also include a call site edge from the variable node and from the function node in E. We omit a similar variant of the case (6). The remaining cases (3)-(5) are the same, except that bind has the additional c parameter and recursive calls always set it to $\perp$.

Finally, the update function in Figure 4 also needs to be updated to store the newly created var nodes. This is done by adding the following single case:

$$\begin{array}{l} \Delta_i(\mathsf{var}(x), [(v, \mathsf{callsite}(m,i))]) = \mathsf{var}(x,s) \\ \qquad \text{for all } \mathsf{var}(x,s) \in V \\ \qquad \text{such that } (\mathsf{var}(x,s), v, \mathsf{callsite}(m,i)) \in E \end{array}$$

5.3 Type checking over dependency graphs

The type system for The Gamma supports a number of primitive types (such as integers and strings) written as α. Composed types include functions and objects with members. Objects are be provided by type providers, but we omit the details here. Types of object members are written as σ and can have multiple arguments:

$$\begin{array}{lcll} \tau & \in & \alpha \mid \tau \to \tau \mid \{m_1 : \sigma_1, \ldots, m_n : \sigma_n\} & (\textit{Types}) \\ \sigma & \in & (\tau_1, \ldots, \tau_n) \to \tau & (\textit{Members}) \end{array}$$

The typing judgements are written in the form $\mathsf{v} \vdash \tau$. They are parameterised by the dependency graph (V, E), but this is not modified during type checking so we keep it implicit rather than writing e.g. $\mathsf{v} \vdash_{(V,E)} \tau$.

Type checking. The typing rules are shown in Figure 8. Types of primitive values n are obtained using a global lookup table Σ (*val*). When type checking a member call (*mem*), we find its dependencies v_i and check that the first one (instance) is an object with the required member m. The types of input arguments of the member then need to match the types of the remaining (non-instance) nodes.

Type checking a function (*fun*) and a variable (*var*) is similar. In both cases, we follow the callsite edge to find the member that accepts the function as an argument. We obtain the type of the function from the type of the i^{th} argument of the member. We use the input type as the type of variable (*var*). For functions, we also check that the resulting type matches the type of the body (*fun*).

Caching results. Performing type checking over the dependency graph, rather than over the abstract syntax tree, enables us to reuse the results of previously type checked parts of a program. As when caching evaluated previews (Section 4), we build a lookup table mapping graph nodes to types. When type checking a node, we first check the cache and, only if it is new, follow the $\vdash$ relation to obtain the type.

As a result, code can be type checked on-the-fly during editing, even when asynchronous type providers are used, and the programmer gets instant feedback without delays.

6 Properties of live coding environment

The dependency graph makes it possible to cache partial results when evaluating previews. The mechanism needs to satisfy two 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, we show that the mechanism is correct and implements a useful optimization.

6.1 Modeling expression evaluation

In The Gamma language, computations are expressed using member access, written as $e.m(e_1, \ldots, e_n)$. In this paper, we do not define how member access evaluates. This has been done elsewhere [23], but more importantly, the evaluation of previews does not rely on the exact specifics of the evaluation, provided that the language satisfies certain basic conditions. The following definitions provides the necessary structure for discussing correctness of previews.

Partial evaluation may reduce an expression under λ-abstraction. We do not require that the reduction of the host language does this. Instead, we define an extended reduction relation and use that in the proofs. The host language only needs to compose well with such extended reduction as captured by the *compositionality* property below. We also require that the language allows elimination of let bindings.

Definition 1 (Host language)**.** Given a relation on expressions $e_1 \rightsquigarrow e_2$ that models small-step evaluation, we define:

- A *preview evaluation context* (also referred to as *context*):

$$\begin{aligned} C[-] \;=\; & \mathsf{let}\ x = -\ \mathsf{in}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ - \mid \lambda x \to - \\ & e_0.m(e_1, \ldots, e_{k-1}, -, e_{k+1}, \ldots e_n) \end{aligned}$$

- An extended reduction relation $\rightsquigarrow_\beta$ such that, for any context C, $C[e_1] \rightsquigarrow_\beta C[e_2]$ whenever $e_1 \rightsquigarrow e_2$.
- Let elimination $\rightsquigarrow_{\mathsf{let}}$ such that, using capture-avoiding substitution, $C[\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2] \rightsquigarrow_{\mathsf{let}} C[e_2[x \leftarrow e_1]]$

We say that $\rightsquigarrow$ is a suitable *host language reduction* if:

- It satisfies the *compositionality* property, that is if $e \rightsquigarrow e'$ and $C[e] \rightsquigarrow_\beta e''$ then also $C[e'] \rightsquigarrow_\beta e''$.
- Let elimination does not affect the result, i.e. if $e \rightsquigarrow_{\mathsf{let}} e'$ and $e' \rightsquigarrow_\beta e''$ then also $e \rightsquigarrow_\beta e''$

The host language in The Gamma is a simple call-by-value functional language without side-effects, and so it satisfies both compositionality and allows let bindings to be eliminated, although the latter affects the performance. The mechanism for preview evaluation presented here would also work for call-by-name languages, but it would suffer from the expected difficulties in the presence of side-effects or non-determinism.

6.2 Correctness of previews

To show that the evaluated previews are correct, we prove two properties. Correctness (Theorem 4) guarantees that, 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. Determinacy (Theorem 5) guarantees that if we cache a preview for a graph node and update the graph, the preview we would evaluate using the updated graph would be the same as the cached preview.

To simplify the proofs, we consider expressions without let bindings. This is possible, because eliminating let bindings does not change the result in the host language (Definition 1) and it also does not change the constructed dependency graph as shown in Lemma 1.

Lemma 1 (Let elimintion)**.** *Given an expression e_1 such that $e_1 \rightsquigarrow_{\mathsf{let}} e_2$ and a lookup table Δ_0 then if $v_1, (V_1, E_1) = \mathsf{bind}_{\emptyset,\Delta_0}(e_1)$ and $v_2, (V_2, E_2) = \mathsf{bind}_{\emptyset,\Delta_1}(e_2)$ such that $\Delta_1 = \mathsf{update}_{V_1,E_1}(\Delta_0)$ then it holds that $v_1 = v_2$ and also $(V_1, E_1) = (V_2, E_2)$.*

Proof. Assume $e_1 = C[\mathsf{let}\ x = e'\ \mathsf{in}\ e'']$ and the resulting $e_2 = C[e''[x \leftarrow e']]$. The case $\mathsf{bind}_{\Gamma,\Delta}(\mathsf{let}\ x = e'\ \mathsf{in}\ e'')$ when binding e_1 is handled using (3).

When binding e_1, the node resulting from binding e' is added to the graph V_1, E_1 and is referenced each time x is used. When binding e_2, the node representing e' is a primitive value, or already present in Δ_1 (added by $\mathsf{update}_{V_1,E_1}$) and is reused each time $\mathsf{bind}_{\Gamma,\Delta_1}(e')$ is called. □

The Lemma 1 provides a way of removing let bindings from an expression, such that the resulting dependency graph remains the same. Here, we bind the original expression first, which adds the node for e' to Δ. In our implementation, this is not needed because Δ is updated while the graph is being constructed using bind. To keep the formalisation simpler, we separate the process of building the dependency graph and updating Δ.

Now, we can show that, given a let-free expression, the preview obtained using a correctly constructed dependency graph is the same as the one we would obtain by directly evaluating the expression. This requires a simple auxiliary lemma and the full proof is shown in Appendix A.

Lemma 2. *[Lookup inversion] Given Δ obtained using* update *as defined in Figure 4 then:*

- *If $v = \Delta(\mathsf{fun}(x), [(v_0, l_0)])$ then $v = \mathsf{fun}(x, s)$ for some s.*
- *If $v = \Delta(\mathsf{mem}(m), [(v_0, l_0), \ldots, (v_n, l_n)])$ then $v = \mathsf{mem}(m, s)$ for some s.*

Proof. By construction of Δ in Figure 4. □

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 $v, (V, E) = \mathsf{bind}_{\emptyset,\Delta}(e)$. If $v \Downarrow d$ over a graph (V, E) then $d = p$ for some p and $e \rightsquigarrow_\beta p$.*

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

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. □

The correctness theorem combines the previous two results.

Theorem 4 (Correctness). *Consider an expression e_1 that has no free variables together with a lookup table Δ_1 obtained from any sequence of expressions according to Figure 4 and e_2 such that $e_1 \rightsquigarrow_\beta e_2$ and let $v_1, (V_1, E_1) = \mathsf{bind}_{\emptyset,\Delta_1}(e_1)$.*

Let $\Delta_2 = \mathsf{update}_{V_1,E_1}(\Delta_1)$ and $v_2, (V_2, E_2) = \mathsf{bind}_{\emptyset,\Delta_2}(e_2)$. If $v_2 \Downarrow d$ over a graph (V_2, E_2) then $d = p$ for some p and $e \rightsquigarrow_\beta p$.

Proof. Direct consequence of Lemma 1 and Theorem 3. □

As discussed above when introducing Lemma 1, in our implementation, Δ is updated during the recursive binding process and so a stronger version of the property holds – namely, $e \rightsquigarrow_\beta p$ for a p that is obtained by calculating preview over a graph obtained directly for the original expression e. We note that this is the case, but do not show it formally to keep aid the clarity of our formalisation.

The second important property that guarantees the correctness of previews shown by the user in our implementation is determinacy. This makes it possible to cache the previews evaluated using $\Downarrow$ using the corresponding graph node as a lookup key.

Theorem 5 (Determinacy). *Let $\Delta_1 = \emptyset$, for any e_1, e_2, assume that the first expression is bound, i.e. $v_1, (V_1, E_1) = \mathsf{bind}_{\emptyset,\Delta_1}(e_1)$, the graph node cache is updated $\Delta_2 = \mathsf{update}_{V_1,E_1}(\Delta_1)$ and a new expression is bound, i.e. $v_2, (V_2, E_2) = \mathsf{bind}_{\emptyset,\Delta_2}(e_2)$. Now, for any v, if $v \Downarrow p$ over (V_1, E_1) then also $v \Downarrow p$ over (V_2, E_2).*

Proof. By induction over $\Downarrow$ over (V_1, E_1), we show that the same evaluation rules also apply over (V_2, E_2).

This is the case, because new graph nodes added to Δ_2 by $\mathsf{update}_{V_1,E_2}$ are only ever added as new nodes in $\mathsf{bind}_{\emptyset,\Delta_2}$ and so the existing nodes and edges of (V_1, E_1) used during the evaluation are unaffected. □

The mechanism used for caching previews, as discussed at the end of Section 4, keeps a preview or a partial preview d in a lookup table indexed by nodes v. The Theorem 5 guarantees that this is a valid strategy. As we update dependency graph during code editing, previous nodes will continue representing the same sub-expressions.

1. Let introduction A. The expression $C_1[C_2[e]]$ is changed to $C_1[\mathsf{let}\ x = e\ \mathsf{in}\ C_2[x]]$ via semantically non-equivalent expression $C_1[C_2[x]]$ where x is unbound variable.

2. Let introduction B. The expression $C_1[C_2[e]]$ is changed to $C_1[\mathsf{let}\ x = e\ \mathsf{in}\ C_2[x]]$ via $C_1[\mathsf{let}\ x = e\ \mathsf{in}\ C_2[e]]$ where x is unused variable.

3. Let elimination A. The expression $C_1[\mathsf{let}\ x = e\ \mathsf{in}\ C_2[x]]$ is changed to $C_1[C_2[e]]$ via semantically non-equivalent expression $C_1[C_2[x]]$ where x is unbound variable.

4. Let elimination B. The expression $C_1[\mathsf{let}\ x = e\ \mathsf{in}\ C_2[x]]$ is changed to $C_1[C_2[e]]$ via $C_1[\mathsf{let}\ x = e\ \mathsf{in}\ C_2[e]]$ where x is unused variable.

5. Editing a non-dependency in let. Assuming $x \notin FV(e_2)$, the expression $C_1[\mathsf{let}\ x = e_1\ \mathsf{in}\ C_2[e_2]]$ changes to an expression $C_1[\mathsf{let}\ x = e_1'\ \mathsf{in}\ C_2[e_2]]$. The preview of a sub-expression e_2 is not recomputed.

6. Editing a non-dependency in a chain. The expression $C[e.m(e_1, \ldots, e_n).m'(e_1', \ldots, e_k')]$ is changed to an expression $C[e.m(e_1, \ldots, e_n).m''(e_1'', \ldots, e_k'')]$. The preview of a sub-expression $e.m(e_1, \ldots, e_n)$ is not recomputed.

Figure 9. Code edit operations that enable preview reuse

6.3 Reuse of previews

In the motivating example in Section 1, the programmer first extracted a constant value into a let binding and then modified a parameter of the last method call in a call chain. We argued that the live coding environment should reuse partially evaluated previews for these two cases. In this section, we prove that this is, indeed, the case in our system.

Figure 9 shows a list of six code edit operations where a preview of the expression (cases 1-4), or a sub-expression (cases 5-6), can be reused. This is the case, because the graph nodes that are bound to the sub-expression before and after the code is changed are the same and hence, a cached preview (stored using the graph node as the key) can be reused.

In some of the operations (cases 1 and 3), the code is changed via an intermediate expression that is semantically different and has only partial preview. This illustrates a typical way of working with code in a text editor using cut and paste operations. Cases 1 and 3 illustrate how our approach allows this way of editing code.

Finally, it is worth noting that our list is not exhaustive. In particular, cases 1-4 only cover let bindings where the bound variable is used once. However, previews can also be reused if the variable appears multiple times.

Lemma 6 (Binding sub-expressions). *Given any Δ_1 together with $e_1 = C[C_1[e]]$ and $e_2C[C_2[e]]$, such that all free variables of e are bound in C, assume that the first expression is bound, i.e. $v_1, (V_1, E_1) = \mathsf{bind}_{\emptyset,\Delta_1}(e_1)$, the graph node cache is updated*

$\Delta_2 = \mathsf{update}_{V_1,E_1}(\Delta_1)$ *and the second expression is bound, i.e.* $v_2, (V_2, E_2) = \mathsf{bind}_{\emptyset,\Delta_2}(e_2)$.

Now, assume $v, G = \mathsf{bind}_{\Gamma_1,\Delta_1}(e)$ *and* $v', G' = \mathsf{bind}_{\Gamma_2,\Delta_2}(e)$ *are the recursive calls to bind e during the first and the second binding, respectively. Then, the graph nodes assigned to the sub-expression e are the same, i.e.* $v = v'$.

Proof. First, assuming that $\forall x \in FV(e).\Gamma_1(x) = \Gamma_2(x)$, we show by induction over the binding process of e when binding $C[C_1[e]]$ that the result is the same. In cases (1) and (6), the updated Δ_2 contains the required key and so the second binding proceeds using the same case. In cases (2) and (7), the second binding reuses the node created by the first binding using case (1) and (6), respectively. Cases (4) and (5) are the same and case (3) follows directly via induction.

Second, when binding let bindings in $C[-]$, the initial $\Gamma = \emptyset$ during both bindings and so the nodes added to Γ_1 and Γ_2 are the same. C_1 and C_2 do not add any new nodes used in e to Γ_1 and Γ_2 and so $v = v'$ using the above. □

Theorem 7 (Preview reuse)**.** *Given the sequence of expressions as specified in Figure 9, if the expressions are bound in sequence and graph node cache updated as specified in Figure 4, then the graph nodes assigned to the specified sub-expressions are the same.*

Proof. Consequence of Lemma 6, using appropriate contexts. In cases with intermediate expressions (1)-(4), binding the intermediate expression introduces additional nodes to Δ, but those are ignored when binding the final expression. □

6.4 Properties of type checking

As noted in Section 5, the focus of this paper is on live previews, but we also use the method based on reusing nodes in a dependency graph for type checking. We do not discuss properties of type checking in detail, but we briefly note how the different properties of live previews extend to corresponding properties of type checking.

Type checking result reuse. In Section 6.3, we show that certain source code edits do not cause the recomputation of previews for the whole expression or a sub-expression. The edits are given in Figure 9. The proof uses the fact that the newly bound graph (after code edit) reuses nodes of the previous graph. This implies that type checking results can be reused in exactly the same way as live previews – they are also stored in a lookup table with graph nodes as keys.

Correctness. The correctness property (Theorem **??**) shows that graph-based preview evaluation matches direct evaluation of expressions. To show a corresponding property for type checking, we would need to provide ordinary type system based on the structure of the expression and prove that the two are equivalent. In our implementation, we only use the presented graph-based type checking method, so we do not provide an alternate account in this paper.

Determinacy. The determinacy property (Theorem 5) guarantees that previews can be cached, because evaluating them again, using $\Downarrow$ over an updated graph, would yield the same result. The same property holds for $\vdash$, meaning that type checking results can be cached. Although the Theorem 5 talks explicitly about $\Downarrow$, it can be easily extended for $\vdash$, because the proof depends on how the graph is updated using $\mathsf{update}_{V,E}$ and the binding process.

7 Design lessons

The motivation for the presented work, briefly outlined in Section 2, is to build a simple data exploration environment that would allow non-experts, like data journalists, transparently work with data. In this paper, we focused on providing live coding experience, which is one important step toward the goal. However, the language we use is a mix of established object-oriented (member access) and functional (function values) features with type providers.

If we were to design a new programming language, there are lessons we can learn from the cases that make type checking and preview evaluation in this paper difficult. This section briefly considers those.

Functions and type providers. When using type providers in a nominally-typed language, the provided types are named, but the names are typically not easy to type [24]. This is not a problem in typical usage where provided members are accessed via dot. Using the worldbank example from Section 5.1, we can access population of two countries using:

```
worldbank.byCountry.'United Kingdom'.
  Indicators.'Population (total)'
```

```
worldbank.byCountry.'Czech Republic'.
  Indicators.'Population (total)'
```

However, the fact that the provided types do not have nice names becomes a problem when want to extract code to access population into a function:

```
let getPopulation c =
  c.Indicators.'Population (total)'
```

Here, the compiler cannot infer the type of c from usage and so we are required to provide a type annotation using an automatically generated name.

Functions and live previews. Providing live previews in a language with ordinary functions suffers from the same problem as type checking of functions.

Our live preview evaluation, discussed in Section 4, can obtain only a delayed preview for the body of getPopulation. The delayed preview we would obtain in this case is $[\![c.\text{Indicators.'Population (total)'}]\!]_c$.

If we know the type of c, we can provide a user interface that lets the user specify a value for c (or, more generally, free variables of the preview) and then evaluate the preview, but it is difficult to provide a meaningful preview automatically.

Wormhole abstractions. In data science scripting, we start with a concrete example and then turn code into a reusable function. This pattern could be supported by the language in a way that makes type checking and preview evaluation easier. Using an imaginary notation, we could write:

```
let uk = worldbank.byCountry.' United Kingdom'

def getPopulation =
  [country : uk].Indicators.'Population (total)'

getPopulation worldbank.byCountry.China
getPopulation worldbank.byCountry.India
```

We tentatively call this notation *wormhole* abstraction and we intend to implement it in future prototypes of The Gamma. The second line is an expression that accesses the population of the UK, using a concrete data source as the input, but it also defines a named function getPopulation that has a parameter country. In a way, we are providing *type annotation* by example, together with a *value annotation* that can be used for live previews.

This way of constructing abstractions is perhaps more akin to how spreadsheets are used – we often write a formula using a concrete cell and then use the "drag down" operation to extend it to other inputs.

8 Related and future work

This paper approaches the problem of live coding environments from a theoretical programming language perspective with a special focus on tooling for data science. Hence, the related and future work spans numerous areas.

Design and human-computer interaction. From a design perspective, the idea of live programming environments has been popularised by Bret Victor [34]. Active research on novel forms of interaction happens in areas such as live coded music [1, 25]. The idea of live previews can be extended to direct manipulation [27]. The Gamma provides limited support for directly manipulating data (Section 2), but we intend to explore this direction further.

Data science tooling. An essential tool in data science is REPL (read-eval-print-loop) [8], which is now widely available. This has been integrated with rich graphical outputs in tools such as Jupyter notebooks [15, 22], but such previews are updated using an explicit command. Integrating our work with Jupyter to provide instant live previews for R or Python would be an interesting extension of the presented work.

Live coding and live previews. Live previews have been implemented in LightTable [11] and, more recently, in editors such as Chrome Developer Tools, but neither presents a simple description of their inner workings. An issue that attracts much attention is keeping state during code edits [2, 17]. This would be an interesting problem if we extended our work to event-based reactive programming.

Structured editing. An alternative approach to ours is to avoid using text editors. Structured editors [31] allow the user to edit the AST and could, in principle, recompute previews based on the performed operations, or preview evaluation as in interactive functional programming [21]. A promising direction is using bi-directional lambda calculus [20]. Finally, abandoning text also enables building richer, more human-centric abstractions as illustrated by Subtext [5]. Our current focus, however, remains on text-based editors.

Dependency analysis. Our use of dependency graphs [16] is first-order. Building dependency graphs involving function calls using modern compiler methods [14] or program slicing [35] would allow us to deduce possible inputs for functions and use those for previews rather than changing the language as suggested in Section 7. This direction is worth considering, but it requires more empirical usability testing.

Semantics and partial evaluation. The evaluation of previews can be seen as a form of partial evaluation [4], done in a way that allows reuse of results. This can be done implicitly or explicitly in the form of multi-stage programming [32]. Both can provide useful perspective for formally analysing how previews are evaluated. Semantically, the evaluation of previews can be seen as a modality [6] and delayed previews are linked to contextual modal type theory [19], which, in turn, can be understood in terms of comonads [9]. This provides an intriguing direction for rigorous analysis of the presented system.

9 Summary

We present The Gamma, a live coding environment for data exploration. The environment bridges the gap between spreadsheets and scripting – live previews give users rapid feedback, while the final result is a fully reproducible script.

In this paper, we focus on the challenge of efficiently providing live previews and type checking code during editing in a free-form text editor. This is a challenge, because users can perform arbitrary text transformations and we cannot recompute previews after each edit.

The key trick is to separate the process into fast *binding phase*, which constructs a dependency graph and slower *evaluation phase* and *type checking phase* that can cache results, using the nodes from the dependency graph created during binding as keys. This makes it possible to quickly parse updated code, reconstruct dependency graph and compute preview using previous, partially evaluated, results.

We describe our approach formally, which serves two purposes. First, we aim to provide easy to use foundations for the growing and important trend of text-based live coding environments. Second, we explore the properties of our system and prove that our method does not recompute previews in a number of common cases and, at the same time, the optimisation still produces correct previews.

References

[1] Samuel Aaron 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*. ACM, 35–46.

[2] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, and Nikolai Tillmann. 2013. It's Alive! Continuous Feedback in UI Programming. In *PLDI*. ACM SIGPLAN.

[3] David Raymond Christiansen. 2013. Dependent type providers. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*. ACM, 25–34.

[4] Charles Consel and Olivier Danvy. 1993. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 493–501.

[5] Jonathan Edwards. 2005. Subtext: uncovering the simplicity of programming. *ACM SIGPLAN Notices* 40, 10 (2005), 505–518.

[6] Matt Fairtlough, Michael Mendler, and Eugenio Moggi. 2001. Special issue: Modalities in type theory. *Mathematical Structures in Computer Science* 11, 4 (2001), 507–509.

[7] Roy Fielding. 2000. Representational state transfer. *Architectural Styles and the Design of Netowork-based Software Architecture* (2000), 76–85.

[8] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. 2002. DrScheme: A programming environment for Scheme. *Journal of functional programming* 12, 2 (2002), 159–182.

[9] Murdoch J Gabbay and Aleksandar Nanevski. 2013. Denotation of contextual modal type theory (CMTT): Syntax and meta-programming. *Journal of Applied Logic* 11, 1 (2013), 1–29.

[10] Marco Gaboardi, Shin-ya Katsumata, Dominic A Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading.. In *ICFP*. 476–489.

[11] Chris Granger. 2012. LightTable: A new IDE concept. *http://www.chris-granger.com/2012/04/12/light-table-a-new-ide-concept/* (2012).

[12] Jonathan Gray, Lucy Chambers, and Liliana Bounegru. 2012. *The data journalism handbook: how journalists can use data to improve the news.* " O'Reilly Media, Inc.".

[13] P Guo. 2013. Data science workflow: Overview and challenges. *blog CACM, Communications of the ACM* (2013).

[14] Ken Kennedy and John R Allen. 2001. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc.

[15] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks-a publishing format for reproducible computational workflows.. In *ELPUB*. 87–90.

[16] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. 1981. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 207–218.

[17] Sean McDirmid. 2007. Living it up with a live programming language. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 623–638.

[18] Alan Mycroft, Dominic Orchard, and Tomas Petricek. 2016. Effect systems revisitedâĂŤcontrol-flow algebra and semantics. In *Semantics, Logics, and Calculi*. Springer, 1–32.

[19] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)* 9, 3 (2008), 23.

[20] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 86–99.

[21] Roland Perera. 2013. *Interactive functional programming.* Ph.D. Dissertation. University of Birmingham.

[22] Fernando Pérez and Brian E Granger. 2007. IPython: a system for interactive scientific computing. *Computing in Science & Engineering* 9, 3 (2007).

[23] Tomas Petricek. 2017. Data Exploration through Dot-driven Development. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[24] Tomas Petricek, Gustavo Guerra, and Don Syme. 2016. Types from data: Making structured data first-class citizens in F. *ACM SIGPLAN Notices* 51, 6 (2016), 477–490.

[25] Charles Roberts, Matthew Wright, and JoAnn Kuchera-Morin. 2015. Beyond editing: extended interaction with textual code fragment. In *NIME*. 126–131.

[26] Peter Sestoft. 2012. *Spreadsheet technology.* Technical Report. Citeseer.

[27] Ben Shneiderman. 1981. Direct manipulation: A step beyond programming languages. In *ACM SIGSOC Bulletin*, Vol. 13. ACM, 143.

[28] Don Syme. 2006. Leveraging. net meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*. ACM, 43–54.

[29] Donald Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. 2013. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of the 2013 workshop on Data driven functional programming*. ACM, 1–4.

[30] Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# asynchronous programming model. *Practical Aspects of Declarative Languages* (2011), 175–189.

[31] Gerd Szwillus and Lisa Neal. 1996. *Structure-based editors and environments.* Academic Press, Inc.

[32] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science* 248, 1 (2000), 211–242.

[33] Mads Torgersen and Neal Gafter. 2012. C Language Specification. *https://github.com/dotnet/csharplang/tree/master/spec, Retrieved 2017* (2012).

[34] Bret Victor. 2012. Learnable programming: Designing a programming system for understanding programs. *http://worrydream.com/LearnableProgramming* (2012).

[35] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

A Appendix

Theorem 0 (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 $e \rightsquigarrow_\beta p$.*

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

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 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, $e_i \rightsquigarrow_\beta p_i$ and so, using compositionality of $\rightsquigarrow$, $e_0.m(e_1, \ldots, e_n) \rightsquigarrow_\beta p$.

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

(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 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 $e_b \rightsquigarrow_\beta p_b$ and so by definition also $\lambda x.e_b \rightsquigarrow_\beta \lambda x.p_b$.

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 context $C[-]$, it holds that $C[e] \rightsquigarrow_\beta C[e_b]$. Using a context $C[-] = \lambda x.-$ it holds that $\lambda x.e \rightsquigarrow_\beta \lambda x.e_b$.

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 context $C[-]$, it holds that $C[e] \rightsquigarrow_\beta C[e_b]$ and for any context $C'[-]$, by definition of $\rightsquigarrow_\beta$ also $C'[\lambda x.e] \rightsquigarrow C'[\lambda x.e_b]$. $\square$