Live coding

ANONYMOUS AUTHOR(S)

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 [Sestoft 2012], 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 [Omar et al. 2017; Szwillus and Neal 1996] 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

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

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 **??**), 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 5) captures the essence of the new perspective.
- We formalise the evaluation of previews (Section 6) and prove that our evaluation and caching mechanism is produces correct previews (Section 7.1) and can effectively reuse partial results (Section 7.2).
- We follow the same method to implement a type checker (Section 8) for our language that is capable of reusing previous results. This makes it possible to efficiently support asynchronously provided types (Section 8.1).
- In more speculative conclusions (Section 9), 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.

2 UNDERSTANDING HOW DATA SCIENTISTS WORK

Data scientists often use general-purpose programming languages such as Python, but the kind of code they write and the way they interact with the system is very different from how software engineers work. This paper focuses on simple data wrangling and data exploration as done, for example, by journalists analysing government datasets. This section illustrates how such data analyses look and provides a justification for the design of our data exploration calculus.

2.1 Data wrangling in Jupyter

Data analysts increasingly use notebook systems such as Jupyter or R Markdown, which make it possible to combine text, equations and code with results of running the code such as tables or visualizations. Notebooks blur the distinction between development and execution. Data analysts write small snippets of code, run them to see results immediately and then revise them.

Jupyter notebooks are used by a variety of users ranging from scientists who implement complex models to journalists who load data, perform simple aggregations and create visualizations. Our initial focus is on simple use cases, such as those of journalists. Tooling that makes notebook systems more interactive and spreadsheet-like in such cases is of crucial importance as it enables more people to produce reproducible and transparent data analyses.

As an example, consider the anlysis done by the Financial Times for an article on plastic waste. The analysis joins datasets from Eurostat, UN Comtrade and more, aggregates the data and builds a visualization comparing plastic waste flows in 2017 and 2018. Figure 1 shows an extract from one notebook from the data analysis. The code of the analysis has a number of remarkable properties:

UN Comtrade exports data

```
import pandas as pd
material = 'plastics' # 'plastics', 'paper'
```

Loading exports data

```
df_mat = pd.read_csv('{material}-2017.csv').fillna(0).sort_values(['country_name', 'period'])
df_mat.head()
```

	period	country_name	kg	country_code
0	2017-01-01	Algeria	43346.0	12
1	2017-03-01	Algeria	32800.0	12
2	2017-03-01	Antigua and Barbuda	17000.0	28

Join to country codes

```
# Set keep_default_na because the Namibia has ISO code NA
df_isos = pd.read_excel('iso.xlsx', keep_default_na=False).drop_duplicates('country_code')
df = df_mat.copy().merge(df_isos, 'left', 'country_code').rename({ 'iso2': 'country_code' }, axis=1)
df.head()
```

	period	country_name	kg	country_code
0	2017-01-01	Algeria	43346.0	DZ
1	2017-03-01	Algeria	32800.0	DZ
2	2017-03-01	Antigua and Barbuda	17000.0	AG

Fig. 1. An excerpt from a Jupyter notebook by Financial Times reporters, which loads trade data from the UN trade database and adds ISO country codes.

- There is no abstraction. The full analysis uses lambda functions as arguments to library calls, but it does not define any top-level functions. Parameterization is done by defining a list of inputs and then having a for loop, or by defining a top-level variable and setting its value. In our example, material is set to "plastics" and a comment lists other options. This lets the analyst easily see and check the results of intermediate steps of the analysis.
- The code is structured as a sequence of commands. Some commands define a variable, either by loading data, or by transforming data loaded previously. Even in Python, data is often treated as immutable. Other commands produce an output that is displayed in the notebook.
- There are many corner cases, such as the fact that the keep_default_na parameter needs to be set to handle Namibia correctly. These are discovered interactively by writing and running a code snippet and examining the output, so providing a rapid feedback is essential.

Many Jupyter notebooks are more complex than the example discussed here. They might define helper functions or even structure code using object-oriented programming. However, simple data analyses such as the one discussed here are frequent enough to pose an interesting and important domain for programming tools research.

2.2 Dot-driven data exploration in The Gamma

Data exploration of the kind discussed in the previous section has been the motivation for a recent programming environment called The Gamma. The scripting language in The Gamma captures the structure discussed above.

A script is a sequence of commands that can either define a variable or produce an output. The language does not support top-level function declarations, although we propose an extension that provides a simple abstraction mechanism and is compatible with providing live previews in Section X. Lambda functions can be used only as method arguments.

Given the limited expressiveness of The Gamma, libraries for it are implemented in other languages, such as JavaScript. More importantly, The Gamma supports type providers, which

```
olympics
  .'filter data'.'Games is'.'Rio (2016)'.then
  .'group data'.'by Athlete'
    .'count distinct Event'
    .su
      count distinct Gender
      count distinct Gold
      count distinct Medal
      count distinct Silver
      count distinct Sport
      count distinct Team
      count distinct Year
      sum Bronze
      sum Gold
      sum Silver
      sum Year
      then
```

(a) The Gamma script to aggregate Olympic medals. We select Rio 2016 games and count the number of distinct events per athlete. We then type '.' to choose further aggregation operations.

```
olympics
  .'filter data'.'Games is'.'Rio (2016)'.then
  .'group data'.'by Athlete'
    .'count distinct Event'.'sum Gold'.then
  .'sort data'.'by Gold descending'.then
  .paging.take(5)
```

Athlete	Event	Gold
Michael Phelps	6	5
Katie Ledecky	5	4
Simone Biles	5	4
Katinka Hosszu	4	3
Usain Bolt	3	3

(b) Live preview in The Gamma programming environment. The table is produced as the data anlyst types and updates on-the-fly to show result at the current cursor position.

Fig. 2. TBD

can be used to generate types, on the fly, for accessing and exploring external data sources. Type providers provide object types with members and The Gamma makes using those convenient by providing auto-complete when the user types the dot symbol ('.') to access a member.

The combination of type providers and auto-complete on "." makes it possible to complete a large number of data exploration tasks through a very simple interaction of selecting operations from a list. An example in Figure 2a summarizes data on Olympic medals using a type provider. The names of identifiers such as 'sum Bronze' do not have any special meaning – they are merely members generated by the type provider, based on information about the data source. The type provider used in this example generates an object with members for different data transformations, such as 'group data', which return furhter objects with members for specifying transformation parameters, such as selecting the grouping key using 'by Athlete'.

The Gamma is more complex than shown here – it supports methods that can take functions as arguments – but the above example illustrates the fact that non-trivial data exploration can be done using a very simple language.

We complement our theoretical work on foundations of live programming environments for data exploration with an implementation for The Gamma. As discussed in the previous section, the assumptions about structure of code that are explicit in The Gamma are also implicitly present in Python and R data analyses produced by journalists, economists and other users with other than programming background. Consequently, our theoretical work applies to simple data analyses using Python and readers not familiar with The Gamma can assume that we use a small subset of Python.

3 LIVE CODING FOR DATA EXPLORATION

This paper presents the foundations of a live coding environment for data exploration that follows the analysis in the previous section. We consider simple data exploration tasks, such as those done by journalists and we implement our ideas in the editor for TheGamma.

Figure 2b shows a TheGamma script exporing Olympic medal statistics in our live coding environment. Before discussing the technical details, it is worth highlighting a number of important points about our approach.

- **Real-world use cases.** Our work has been motivated by the needs outlined in the previous section. In particular, we have been focusing on a langauge that is simple, yet sufficient for completing basic data exploration tasks that a journalist might want to do. We have used our live coding environment to perform a number of simple data analyses looking at topics such as public government spending, crises in financial markets and activities of a research institute. Qualitative observations about our experience are reported in Section X. Our tools can be used in any modern web browser and are available as an open-source package.
- **Text editor integration.** Our approach is based on an ordinary text editor that allows unrestricted modification of the code. Our parser supports error-recovery and we attempt to give previews whenever the code can be correctly parsed and type-checked up to the current location. Notably, we do not, in general, know how the user modified the code. This is in contrast with approaches based on structured editing where the editor tracks what changes to code have been made. Using an ordinary text editor makes providing live previews harder, but we believe that many users still favor the flexibility of plain text editors.
- **Data stay, code changes.** Our focus has been on the scenario when input data is a static file that does not change, but the data analyst modifies the source code. This is in contrast with work on incremental computation, which focuses on scenarios where code stays the same, but input data change.

Programs, commands, terms, expressions and values

$$
\begin{array}{lll}
p = c_1; \ldots; c_n & t = o \mid x & e = t \mid \lambda x \rightarrow e \\
c = \textsf{let}\ x = t \mid t & \quad \mid t.m(e, \ldots, e) & v = o \mid \lambda x \rightarrow e
\end{array}
$$

Evaluation contexts of expressions

$$
\begin{array}{lll}
C_e[-] & = & C_e[-].m(e_1, \ldots, e_n) \mid o.m(v_1, \ldots, v_m, C_e[-], e_1, \ldots, e_n) \mid - \\
C_c[-] & = & \textsf{let}\ x = C_e[-] \mid C_e[-] \\
C_p[-] & = & o_1;\ \ldots;\ o_k;\ C_c[-];\ c_1;\ \ldots;\ c_n
\end{array}
$$

Let elimination and member reduction

$$
\begin{array}{ll}
o_1;\ \ldots;\ o_k;\ \textsf{let}\ x = o;\ c_1;\ \ldots; c_n \rightsquigarrow & \\
\qquad o_1;\ \ldots;\ o_k;\ o; c_1[x \leftarrow o];\ \ldots; c_n[x \leftarrow o] & \text{(let)}
\end{array}
$$

$$
o.m(v_1, \ldots, v_n) \rightsquigarrow_\epsilon o' \implies C_p[o.m(v_1, \ldots, v_n)] \rightsquigarrow C_p[o'] \qquad \text{(external)}
$$

Fig. 3. Syntax, contexts and reduction rules of the data exploration calculus

That said, many of the methods that we used for evaluating live previews such as maintaining the dependency graph with nodes for individual syntactic elements are closely related to methods used in incremental computation research. We do not attempt to combine the two, but note that this is an interesting future direction.

4 DATA EXPLORATION CALCULUS

The *data exploration calculus* is a small, formally tractable language for data exploration. The calculus is not Turing-complete and it can only be used together with external libraries that define what objects are available and what the behaviour of their members is. This is sufficient to capture the simple data analyses discussed in Section 2. We define the caluclus in this section and then use it as the basis for discussing our live preview mechanism in Section 5.

We initially present the data exploration calculus without a static type system. Our main focus is on the live preview mechanism which can be done equally well without types. We consider type checking in Section 8, because the same mechanism that is used for live previews can also be used for incremental type-checking.

4.1 Language syntax

The calculus combines object-oriented features such as member access with functional features including lambda functions. The syntax is defined in Figure 3. Object values o are defined by external libraries that are used in conjunction with the core calculus.

A program p in the data exploration calculus consists of a sequence of commands c. A command can be either a let binding or a term. Let bindings define variables x that can be used in subsequent commands. The calculus supports lambda functions, but they can only appear as arguments in method calls. A term t can be a value, variable or a member access, while an expression e, which can appear as an argument in member access, can be a lambda function or a term.[1]

4.2 Operational semantics

The data exploration calculus is a call-by-value language. We model the evaluation as a small-step reduction $\rightsquigarrow$. Fully evaluating a program results in an irreducible sequence of objects $o_1;\ \ldots;\ o_n$ (one object for each command, including let bindings) which can be displayed as intermediate results of the data analysis. The operational semantics is parameterized by a relation $\rightsquigarrow_\epsilon$ that models the functionality of the external libraries used with the calculus and defines reduction rules for member accesses. The relation has the following form:

$$o_1.m(v_1, \ldots, v_n) \rightsquigarrow_\epsilon o_2$$

The operation is invoked on an object and takes values (objects or function values) as arguments. It always results in an opaque object. Figure 3 defines the reduction rules in terms of $\rightsquigarrow_\epsilon$ and evaluation contexts; C_e specifies left-to-right evaluation of arguments of a method call, C_c specifies evluation of a command and C_p defines top-to-bottom evaluation of a program. The rule (external) applies reductions provided by an external library in a call-by-value order and (let) substitutes a value of an evaluated variable in all subsequent commands. Note that (let) leaves the value of the variable in the resulting list of commands.

4.3 Properties

The data exploration calculus is a very simple language and it has a number of desirable properties. However, some of those require that the relation $\rightsquigarrow_\epsilon$, which defines evaluation for external libraries, satisfies a number of conditions. Our main motivation is to capture properties that allow us to prove that our method of evaluating live previews, discussed in Section 7, is correct.

Definition 1 (Further reductions)**.** We define two additional reduction relations:

[1] Similar but weaker restrictions on the use of lambda functions exist in other languages. For example, in order to guide type inference, lambda functions in C# can appear as either method arguments or assigned to an explicitly typed variable, but they cannot be assigned to an implicitly typed variable.

- We write $\rightsquigarrow^*$ for the reflexive, transitive closure of $\rightsquigarrow$
- We write $\rightsquigarrow_{\mathsf{let}}$ for a call-by-name let binding elimination
  $c_1;\ \ldots;\ c_{k-1};\ \mathsf{let}\ x = t;\ c_{k+1};\ \ldots; c_n \ \rightsquigarrow_{\mathsf{let}}\ c_1;\ \ldots;\ c_{k-1};\ t; c_{k+1}[x \leftarrow t];\ \ldots; c_n[x \leftarrow t]$

We say that two expressions e and e' are observationally equivalent if, for any context C, the expressions $C[e]$ and $C[e']$ reduce to the same value. Lambda functions $\lambda x \rightarrow 2$ and $\lambda x \rightarrow 1+1$ are not equal, but they are observationally equivalent. We require that external libraries satisfy two conditions. First, when a method is called with observationally equivalent values as arguments, it should return the same value. Second, the evaluation of $o.m(v_1, \ldots, v_n)$ should be defined for all o, i and v_i. The external library will typically satisfy this by defining an error object and reducing all invalid calls to the error object, following the standard method of Milner [1978].

Definition 2 (External library)**.** An external library consists of a set of objects O and a reduction relation $\rightsquigarrow_\epsilon$ that satsifies the following two properties:

- *Completeness.* For all o, m, i and all $v_1, \ldots, v_i$, there exists o' such that $o.m(v_1, \ldots, v_i) \rightsquigarrow_\epsilon o'$.
- *Compositionality.* For observationally equivalent arguments, the reduction returns the same object, i.e. given $e_0, e_1, \ldots, e_n$ and $e'_0, e'_1, \ldots, e'_n$ and m such that $e_0.m(e_1, \ldots, e_n) \rightsquigarrow^* o$ and $e'_0.m(e'_1, \ldots, e'_n) \rightsquigarrow^* o'$ then if for any contexts $C_0, C_1, \ldots, C_n$ it holds that if $C_i[e_i] \rightsquigarrow^* o_i$ and $C_i[e'_i] \rightsquigarrow^* o_i$ for some o_i then $o = o'$.

The compositionality condition is essential for proving the correctness of our live preview mechanism. Completeness allows us to prove normalization, i.e. all programs reduce to a value – although the resulting value may be an error value provided by the external library.

Theorem 1 (Normalization)**.** *For all p, there exists n and $o_1, \ldots, o_n$ such that $p \rightsquigarrow^* o_1; \ldots; o_n$.*

Proof. A program that is not a sequence of values can be reduced and reduction decreases the size of the program. See Appendix A.1 for more detail. □

Although the reduction rules (let) and (external) of the data exploration calculus define an evaluation in a call-by-value order, eliminating let bindings in a call-by-name way using the $\rightsquigarrow_{\mathsf{let}}$ reduction does not affect the result. This simplifies our later proof of live preview correctness in Section 7.1.

Lemma 2 (Let elimination for a program)**.** *Given any program p such that $p \rightsquigarrow^* o_1; \ldots; o_n$ for some n and $o_1, \ldots, o_n$ then if $p \rightsquigarrow_{\mathsf{let}} p'$ for some p' then also $p' \rightsquigarrow^* o_1; \ldots; o_n$.*

Proof. The elimination of let binding transforms a program $c_1;\ \ldots;\ c_{k-1};\ \mathsf{let}\ x = t;\ c_{k+1};\ \ldots; c_n$ to a program $c_1;\ \ldots;\ c_{k-1};\ t; c_{k+1}[x \leftarrow t];\ \ldots; c_n[x \leftarrow t]$. The reduction steps for the new program can be constructed using the steps of $p \rightsquigarrow^* o_1; \ldots; o_n$. The new command t reduces to an object o using the same steps as the original term t in $\mathsf{let}\ x = t$ but with context $C_c = -$ rather than $C_c = \mathsf{let}\ x = -$; the terms t introduced by substitution also reduce using the same steps as before, but using contexts in which the variable x originally appeared. □

5 FORMALISING LIVE CODING ENVIRONMENT

A naive way of providing live previews during code editing would be to re-evaluate the code after each change. This would be wasteful – for example, when writing code, most changes are additive and the preview can be updated by evaluating just the newly added code. In this section, we develop foundations for an efficient way of displaying live previews for the data exploration calculus.

5.1 Maintaining dependency graph

The key idea behind our method is to maintain a dependency graph [Kuck et al. 1981] 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. When binding a new expression to the graph, we reuse previously created nodes as long as they have the same dependencies. For expressions that have a new structure, we create new nodes.

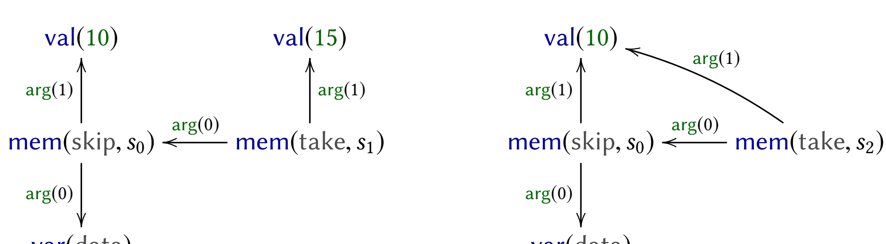
The nodes of the graph serve as unique keys into a lookup table with previously evaluated parts of the program. When a preview is requested for an expression, we use the graph node bound to the expression to find a preview. If a preview has not been evaluated, we force the evaluation of all dependencies in the graph and then evaluate the operation represented by the current node.

5.1.1 Elements of the graph. The nodes of the graph represent individual operations of the computation. In our design, the nodes are used as cache keys, so we attach a unique symbol s 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. Writing s for symbols and i for integers, nodes (vertices) v and edge labels l are defined as:

$$\begin{array}{lcll} v & = & \mathsf{val}(o) \mid \mathsf{var}(x) \mid \mathsf{mem}(m,s) \mid \mathsf{fun}(x,s) & (\textit{Vertices}) \\ l & = & \mathsf{body} \mid \mathsf{arg}(i) & (\textit{Edge labels}) \end{array}$$

The val node represents a primitive value and contains the object 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 s – 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). 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.



(a) Graph constructed from the following initial expression: let x = 15 in data.skip(10).take(x)

(b) Updated graph after changing x to 10: let x = 10 in data.skip(10).take(x)

Fig. 4. Dependency graphs formed by two steps of the live programming process.

5.1.2 Example graph. Figure 4 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 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 symbol s_0 attached to the node remains the same and previously computed previews can be reused. This part of the program is not recomputed. The arg(1) dependency of the take call changed and so we create a new node mem(skip, s_2) with a fresh symbol s_2. The preview for this node is then computed as needed using the already known values of its dependencies.

5.1.3 Reusing graph nodes. The binding process takes an expression and constructs a dependency graph reusing existing nodes when possible. For this, we use a lookup table of member access and function value nodes. The key in the lookup table is formed by a node kind (for disambiguation) together with a list of dependencies. A node kind is a member access containing the member name or a function containing the bound variable name; a lookup table Δ then maps a node kind with a list of dependencies to a cached node:

$$\begin{array}{ll} k \;=\; \mathsf{fun}(x) \mid \mathsf{mem}(m) & (\textit{Node kinds}) \\ \Delta(k, [(v_1, l_1), \ldots, (v_n, l_n)]) & (\textit{Lookup for a node}) \end{array}$$

The example on the second line looks for a node of a kind k that has dependencies $v_1, \ldots, v_n$ labeled with labels $l_1, \ldots, l_n$. We write $\Delta(k, l)\downarrow$ when a value for a given key is not defined. For example, when creating the graph in Figure 4b, 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 mem(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 mem(take, s_2) using a fresh symbol s_2.

5.2 Binding expressions to a graph

After parsing updated code, we update the dependency graph and link each node of the abstract syntax tree to a node of the dependency graph. This process is called binding and is defined by the bind-expr function (Figure 5) and bind-prog function (Figure 6). Both functions are annotated with a lookup table Δ discussed in Section 5.1 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 $\mathsf{bind\text{-}expr}_{\Gamma,\Delta}(e)$ returns a node v corresponding to the expression e paired with a dependency graph (V, 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. The $\mathsf{bind\text{-}prog}_{\Gamma,\Delta}$ function works similarly, but takes a sequence of commands and returns a sequence of nodes.

5.2.1 Binding expressions. When binding member access, we use bind-expr recursively to get a node and a dependency graph for each sub-expression. The nodes representing sub-expressions are then used as dependencies for lookup into Δ, together with their labels. If a node already exists in Δ it is reused (1). Alternativelym, we create a new node containing a fresh symbol (2).

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

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

$$\mathsf{bind\text{-}expr}_{\Gamma,\Delta}(\lambda x \rightarrow e) = v, (\{v\} \cup V, \{e\} \cup E) \tag{3}$$
$$\quad \text{when } \Gamma_1 = \Gamma \cup \{x, \mathsf{var}(x)\} \text{ and } v_0, (V, E) = \mathsf{bind\text{-}expr}_{\Gamma_1,\Delta}(e) \text{ and } v = \Delta(\mathsf{fun}(x), [(v_0, \mathsf{body})])$$
$$\quad \text{let } e = (v, v_0, \mathsf{body})$$

$$\mathsf{bind\text{-}expr}_{\Gamma,\Delta}(\lambda x \rightarrow e) = v, (\{v\} \cup V, \{e\} \cup E) \tag{4}$$
$$\quad \text{when } \Gamma_1 = \Gamma \cup \{x, \mathsf{var}(x)\} \text{ and } v_0, (V, E) = \mathsf{bind\text{-}expr}_{\Gamma_1,\Delta}(e) \text{ and } \Delta(\mathsf{fun}(x), [(v_0, \mathsf{body})])\downarrow$$
$$\quad \text{let } v = \mathsf{fun}(x, s), s \text{ fresh and } e = (v, v_0, \mathsf{body})$$

$$\mathsf{bind\text{-}expr}_{\Gamma,\Delta}(o) = \mathsf{val}(o), (\{\mathsf{val}(o)\}, \emptyset) \tag{5}$$

$$\mathsf{bind\text{-}expr}_{\Gamma,\Delta}(x) = v, (\{v\}, \emptyset) \text{ when } v = \Gamma(x) \tag{6}$$

Fig. 5. Binding rules that define a construction of a dependency graph for an expression.

$$\mathsf{bind\text{-}prog}_{\Gamma,\Delta}(\mathsf{let}\ x = e; c_2; \ldots; c_n) = v_1; \ldots; v_n, (\{v_1\} \cup V \cup V_1, E \cup E_1) \tag{7}$$
$$\quad \text{let } v_1, (V_1, E_1) = \mathsf{bind\text{-}expr}_{\Gamma,\Delta}(e_1) \text{ and } \Gamma_1 = \Gamma \cup \{(x, v_1)\}$$
$$\quad \text{and } v_2; \ldots; v_n, (V, E) = \mathsf{bind\text{-}prog}_{\Gamma_1,\Delta}(c_2; \ldots; c_n)$$

$$\mathsf{bind\text{-}prog}_{\Gamma,\Delta}(e; c_2; \ldots; c_n) = v_1; \ldots; v_n, (\{v_1\} \cup V \cup V_1, E \cup E_1) \tag{8}$$
$$\quad \text{let } v_1, (V_1, E_1) = \mathsf{bind\text{-}expr}_{\Gamma,\Delta}(e) \text{ and } v_2; \ldots; v_n, (V, E) = \mathsf{bind\text{-}prog}_{\Gamma_1,\Delta}(c_2; \ldots; c_n)$$

$$\mathsf{bind\text{-}prog}_{\Gamma,\Delta}([]) = [], (\emptyset, \emptyset) \tag{9}$$

Fig. 6. Binding rules that define a construction of a dependency graph for a program.

If a lambda function 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 unknown 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 (3) or may not (4) be already present in the lookup table.

5.2.2 Binding programs. When binding a program, we bind the first command and then recursively process remaining commands until we reach an empty list of commands (9). For let binding (7), we bind the expression e assigned to the variable to obtain a graph node v_1. We then bind the remaining commands 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. When the command is just an expression (8), we bind the expression using bind-expr.

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

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

$$\Delta_i(\mathsf{fun}(x), [(v_1, \mathsf{body})]) = \mathsf{fun}(x, s)$$
$$\quad \text{when } \mathsf{fun}(x, s) \in V \text{ and } (\mathsf{fun}(x, s), v_1, \mathsf{body}) \in E$$

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

Fig. 7. Updating the node cache after binding a new graph

5.3 Edit and rebind loop

The binding process formalised in Section 5.2 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 $\Delta_0, \Delta_1, \Delta_2, \ldots$ The initial lookup table is empty, i.e. $\Delta_0 = \emptyset$. At a step i, we parse a program p_i (consisting of several commands) and obtain a new dependency graph using the previous Δ. The result is a sequence of nodes corresponding to commands of the program and a graph (V, E):

$$v_1; \ldots; v_n, (V, E) = \mathsf{bind\text{-}prog}_{\emptyset, \Delta_{i-1}}(p_i)$$

The new state of the cache is computed by calling the $\mathsf{update}_{V,E}(\Delta_{i-1})$ function defined in Figure 7. The function adds newly created nodes from the graph (V, E) to the previous cache Δ_{i-1} and returns a new cache Δ_i. We only cache nodes for function and member accesses – nodes for variables and primitive values will remain the same thanks to the way they are constructed.

6 EVALUATING PREVIEWS

The binding process described in the previous section constructs a dependency graph after code changes. The nodes in the dependency graph correspond to individual operations that will be performed when running the program. When evaluating a preview, we attach (partial) results to nodes of the graph. Since the binding process reuses nodes when their dependencies do not change, previews for expressions (or sub-expressions) of a program can be reused when updating a preview.

In this section, we describe how previews are evaluated. The evaluation is done over the dependency graph, rather than over the structure of the program expressions as in the operational semantics given in Section 4.2. We analyse the preview evaluation formally in Section 7 and show that the resulting previews are the same as the result we would get by directly evaluating the code and, also, that no recomputation occurs when code is edited in certain ways.

6.1 Previews and delayed previews

Programs in the data exploration calculus consist of sequence of commands. Those can always be evaluated to a value with a preview that can be displayed to the user. However, we also support previews for all sub-expressions. This can be problematic if the current sub-expression is inside the body of a function. For example:

$$\mathsf{let}\ \mathsf{top} = \mathsf{movies.take}(10).\mathsf{map}(\lambda x \rightarrow x.\mathsf{getReleased}().\mathsf{format}(\text{"dd-mm-yyyy"}))$$

Here, we can directly evaluate sub-expressions movies and $\mathsf{movies.take}(10)$, but not $x.\mathsf{getReleased}()$ and $x.\mathsf{getReleased}().\mathsf{format}(\text{"dd-mm-yyyy"})$ because they contain a free variable x. Our preview evaluation algorithm addresses this by producing two kinds of previews. A *fully evaluated preview* is just a value, while a *delayed preview* is a partially evalauted expression with free variables:

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

A fully evaluated preview p can be either a primitive object or a function value with no free variables. A possibly delayed preview d can be either an evaluated preview p or an expression e that requires variables Γ. For simplicity, we use an untyped language and so Γ is just a list of variables $x_1, \ldots, x_n$.

A delayed preview is not necessarily the body of a lambda function as it appears in the source code. We partially evaluate sub-expressions of the body that do not have free variables or that have free variables bound by an earlier let binding.

Our implementation does not currently display delayed previews to the user, but there is a number of possible approaches for doing that. Most interestingly, since lambda functions always appear as arguments of a member access, we could force the evaluation of the surrounding expression, capture (a number of) values passed as inputs to the lambda function and display a preview based on those. Finally, in Section 9, we also consider a more speculative design for an abstraction mechanism that supports live previews that could, in some cases, replace lambda function.

6.2 Evaluation of previews

The evaluation of previews is defined in Figure 8. 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 nodes V and edges E of the dependency graph are parameters of the $\Downarrow$ relation, but they do not change during the evaluation and so we do not explicity write them.

The auxiliary relation $v \Downarrow_{\mathsf{lift}} d$ 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). A node representing a value is evaluated to a value (val) and a node representing an unbound variable is reduced to a delayed preview that requires the variable and returns its value (var).

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

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

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

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

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

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

Fig. 8. Rules that define evaluation of previews over a dependency graph for a program

For member access, we distinguish two cases. If all arguments evaluate to values (member-val), then we use the evaluation relation defined by external libraries $\rightsquigarrow_\epsilon$ to immediately evaluate the member access and produce a value. If some of the arguments are delayed (member-expr), because the member access is inside the body of a lambda function, 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.

6.3 Caching of evaluated previews

For simplicity, the relation $\Downarrow$ in Figure 8 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 cached 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.

6.4 Theories of delayed previews

The operational semantics presented in this paper serves two purposes. It gives a simple guide for implementing text-based live coding environments for data science and we use it to prove that our optimized way of producing live previews is correct. However, some aspects of our mechanism are related to important work in semantics of programming languages and deserve to be mentioned.

The construction of delayed previews is related to meta-programming. Assuming we have delayed previews $[\![e_0]\!]_x$ and $[\![e_1]\!]_y$ and we invoke a member m on e_0 using e_1 as an argument. To do this, we construct a new delayed preview $[\![e_0.m(e_1)]\!]_{x,y}$. This operation is akin to expression splicing from meta-programming [Syme 2006; Taha and Sheard 2000].

The semantics of delayed previews can be more formally captured by Contextual Modal Type Theory (CMTT) [Nanevski et al. 2008] and comonads [Gabbay and Nanevski 2013]. In CMTT, $[\Psi]A$ denotes that a proposition A is valid in context Ψ, which is similar 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 [Gaboardi et al. 2016; Mycroft et al. 2016]. 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 $[\![A]\!]_\emptyset \to A$.

7 PROPERTIES OF LIVE CODING ENVIRONMENT

The use of a dependency graph when evaluating previews makes it possible to cache partial results through a mechanism discussed in Section 5.1 and Section 6.3. The mechanism satisfies two properties. First, if we evaluate a preview using dependency graph with caching, it is the same as the value we would obtain by evaluating the expression directly. Second, the evaluation of previews using dependency graphs reuses – in some cases – previously evaluated partial results. In other words, we show that the mechanism is correct and implements a useful optimization.

7.1 Correctness of previews

To show that the evaluated previews are correct, we prove two properties. Correctness (Theorem 6) guarantees that, no matter how a dependency graph is constructed, when we use it to evaluate previews for a program, the previews are the same as the values we would obtain by evaluating the program commands directly. Determinacy (Theorem 7) 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 programs without let bindings. This is possible, because eliminating let bindings does not change the result of evaluation, as shown earlier in Lemma 2, and it also does not change the constructed dependency graph as shown below in Lemma 3.

Lemma 3 (Let elimintion for a dependency graph). *Given programs p_1, p_2 such that $p_1 \rightsquigarrow_{\text{let}} p_2$ and a lookup table Δ_0 then if $v_1; \ldots; v_n, (V, E) = \text{bind-prog}_{\emptyset,\Delta_0}(p_1)$ and $v'_1; \ldots; v'_n, (V', E') = \text{bind-prog}_{\emptyset,\Delta_1}(p_2)$ such that $\Delta_1 = \text{update}_{V,E}(\Delta_0)$ then for all i, $v_i = v'_i$ and also $(V, E) = (V', E')$.*

Proof. Assume $p_1 = c_1; \ldots; c_{k-1}; \text{let } x = e; c_{k+1}; \ldots; c_n$ and the let binding is eliminated resulting in $p_2 = c_1; \ldots; c_{k-1}; e; c_{k+1}[x \leftarrow e]; \ldots; c_n[x \leftarrow e]$. When binding p_1, the case $\text{bind-prog}_{\Gamma,\Delta}(\text{let } x = e)$ is handled using (7) and the node resulting from binding e is added to the graph V, E. It is then referenced each time x appears in subsequent commands $c_{k+1}; \ldots; c_n$. When binding p_2, the node resulting from binding e is a primitive value or a node already present in Δ_1 (added by $\text{update}_{V,E}$) and is reused each time $\text{bind-expr}_{\Gamma,\Delta_1}(e)$ is called. □

The Lemma 3 provides a way of removing let bindings from a program, such that the resulting dependency graph remains the same. Here, we bind the original program 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-expr. To keep the formalisation simpler, we separate the process of building the dependency graph and updating Δ and thus Lemma 3 requires an extra binding step.

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.

Lemma 4 (Lookup inversion). *Given Δ obtained using* update *as defined in Figure 7 then:*
- *If $v = \Delta(\text{fun}(x), [(v_0, l_0)])$ then $v = \text{fun}(x, s)$ for some s.*
- *If $v = \Delta(\text{mem}(m), [(v_0, l_0), \ldots, (v_n, l_n)])$ then $v = \text{mem}(m, s)$ for some s.*

Proof. By construction of Δ in Figure 7. □

Theorem 5 (Term preview correctness). *Given a term t that has no free variables, together with a lookup table Δ obtained from any sequence of programs using* bind-prog *(Figure 6) and* update *(Figure 7), then let $v, (V, E) = \text{bind-expr}_{\emptyset,\Delta}(t)$.*
If $v \Downarrow p$ over a graph (V, E) then $p = o$ for some value o and $t \rightsquigarrow^* o$.

Proof. First note that, when combining recursively constructed sub-graphs, the bind-expr function 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 – the newly added nodes and edges do not introduce non-determinism to the rules given in Figure 8.

We prove a more general property showing that for any e, its binding $v, (V, E) = \text{bind-expr}_{\emptyset,\Delta}(e)$ and any evaluation context C such that $C[e] \rightsquigarrow o$ for some o, one of the following holds:

a. If $FV(e) = \emptyset$ then $v \Downarrow p$ for some p and $C[p] \rightsquigarrow o$
b. If $FV(e) \neq \emptyset$ then $v \Downarrow [\![e_p]\!]_{FV(e)}$ for some e_p and $C[e_p] \rightsquigarrow o$

In the first case, p is a value, but it is not always the case that $e \rightsquigarrow^* p$, because p may be lambda function and preview evaluation may reduce sub-expression in the body of the function. Using a context C in which the value reduces to an object avoids this problem.

The proof of the theorem follows from the more general property. Using a context $C[-] = -$, the term t reduces $t \rightsquigarrow^* t' \rightsquigarrow_\epsilon o$ for some o and the preview p is a value o because $C[p] = p = o$. The proof is by induction over the binding process, which follows the structure of the expression. The full proof is shown in Appendix A.2. □

The correctness theorem combines the previous two results.

Theorem 6 (Program preview correctness)**.** *Consider a program $p = c_1; \ldots; c_n$ that has no free variables, together with a lookup table Δ_0 obtained from any sequence of programs using* bind-prog *(Figure 6) and* update *(Figure 7). Assume a let-free program $p' = t_1; \ldots; t_n$ such that $p \rightsquigarrow^*_{\text{let}} p'$.*
Let $v_1; \ldots; v_n, (V, E) = \text{bind-prog}_{\emptyset,\Delta_0}(p)$ and define updated lookup table $\Delta_1 = \text{update}_{V,E}(\Delta_0)$ and let $v'_1; \ldots; v'_n, (V', E') = \text{bind-prog}_{\emptyset,\Delta_1}(p)$.
If $v'_i \Downarrow p_i$ over a graph (V', E') then $p_i = o_i$ for some value o_i and $t_i \rightsquigarrow o_i$.

Proof. Direct consequence of Lemma 3 and Theorem 5. □

As discussed earlier, our implementation updates Δ during the recursive binding process and so a stronger version of the property holds – namely, previews calculated over a graph obtained directly for the original program p are the same as the values of the fully evaluated program. We note that this is the case, but do not show it formally to aid the clarity of our formalisation.

The second important property that guarantees that our mechanism always displays correct previews is determinacy. This makes it possible to cache the previews evaluated using $\Downarrow$ using the corresponding graph node as a lookup key.

Theorem 7 (Program preview determinacy)**.** *For some Δ and for any programs p, p', assume that the first program is bound, i.e. $v_1; \ldots; v_n, (V, E) = \text{bind-prog}_{\emptyset,\Delta}(p)$, the graph node cache is updated $\Delta' = \text{update}_{V,E}(\Delta)$ and the second program is bound, i.e. $v'_1; \ldots; v'_m, (V', E') = \text{bind-prog}_{\emptyset,\Delta'}(p')$. Now, for any v, if $v \Downarrow p$ over (V, E) then also $v \Downarrow p$ over (V', E').*

Proof. By induction over $\Downarrow$ over (V, E), we show that the same evaluation rules also apply over (V', E'). This is the case, because new graph nodes added to Δ' by $\text{update}_{V,E}$ are only ever added as new nodes in $\text{bind-prog}_{\emptyset,\Delta'}$ and so the existing nodes and edges of (V, E) used during the evaluation are unaffected. □

The mechanism used for caching previews, as discussed in Section 6.3, keeps a preview or a partial preview d in a lookup table indexed by nodes v. The Theorem 7 guarantees that this is a valid strategy. As we update dependency graph during code editing, previous nodes will continue representing the same sub-expressions.

7.2 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.

1. Let introduction A. The expression $C_1[C_2[e]]$ is changed to $C_1[\text{let } x = e \text{ in } C_2[x]]$ 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 $C_1[\text{let } x = e \text{ in } C_2[x]]$ via $C_1[\text{let } x = e \text{ in } C_2[e]]$ where x is unused variable.
3. Let elimination A. The expression $C_1[\text{let } x = e \text{ in } C_2[x]]$ 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 $C_1[\text{let } x = e \text{ in } C_2[x]]$ is changed to $C_1[C_2[e]]$ via $C_1[\text{let } x = e \text{ in } C_2[e]]$ where x is unused variable.
5. Editing a non-dependency in let. Assuming $x \notin FV(e_2)$, the expression $C_1[\text{let } x = e_1 \text{ in } C_2[e_2]]$ changes to an expression $C_1[\text{let } x = e_1' \text{ in } C_2[e_2]]$. 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.

Fig. 9. Code edit operations that enable preview reuse

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 8 (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. $v_1, (V_1, E_1) = \text{bind-expr}_{\emptyset,\Delta_1}(e_1)$, the graph node cache is updated $\Delta_2 = \text{update}_{V_1,E_1}(\Delta_1)$ and the second expression is bound, i.e. $v_2, (V_2, E_2) = \text{bind-expr}_{\emptyset,\Delta_2}(e_2)$.*

Now, assume $v, G = \text{bind-expr}_{\Gamma_1,\Delta_1}(e)$ and $v', G' = \text{bind-expr}_{\Gamma_2,\Delta_2}(e)$ 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 9 (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 7, then the graph nodes assigned to the specified sub-expressions are the same.*

Proof. Consequence of Lemma 8, 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. □

8 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 8.1) can make type checking time consuming, and so we use the dependency graph also for type checking (Section 8.3). Type checking lambda functions (Section 8.2) requires a slight extension of the model discussed in Section 5.

8.1 Asynchronously provided types

Data available in The Gamma can be defined using several kinds of type providers. The type provider used in Figure **??** as asynchronous [Syme et al. 2011]. 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 [Syme et al. 2013].

Type providers can also be implemented as REST services [Fielding 2000] 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 5.

8.2 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:

$$\text{movies.sortBy}(\lambda x \rightarrow x.\text{getBudget}())$$

If movies is a collection of Movie objects, the type of the lambda function must be Movie $\rightarrow$ bool and so the type of x is Movie. This is similar to type checking of lambda functions in C# [Torgersen and Gafter 2012], 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 5, 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 10 shows the result of binding $o.m(\lambda x \rightarrow x)$. 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 $\mathsf{arg}(1)$.

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-expr 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:

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

The key parts of the revised definition of the bind-expr function are shown in Figure 11. We now write $\mathsf{bind\text{-}expr}_{\Gamma,\Delta,c}$ 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-expr has the additional c parameter and recursive calls always set it to $\perp$.

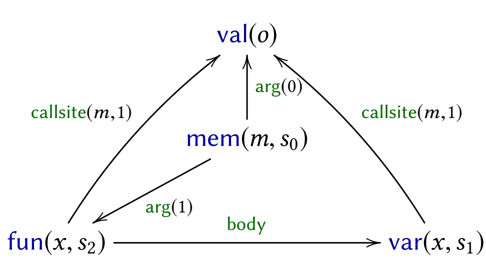


Fig. 10. Dependency graph for $o.m(\lambda x \rightarrow x)$

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

$$\begin{aligned}
&\text{bind-expr}_{\Gamma,\Delta,(v_c,l_c)}(\lambda x\to e)\\
&\quad\text{when } (\text{var}(x),[(v_c,l_c)])\notin\\
&\quad\text{let } v_x=\text{var}(x,s_x), s_x \text{ fres}\\
&\quad\text{and } \Gamma_1=\Gamma\cup\{x,v_x\}\\
&\quad\text{and } v_0,(V_0,E_0)=\text{bind-ex}\\
&\quad\text{when } (\text{fun}(x),[(v_0,\text{body})\\
&\quad\text{let } v=\text{fun}(x,s_f), s_f \text{ fresh}\\
&\quad\text{let } E=\{(v,v_0,\text{body}),(v,
\end{aligned}$$

Fig. 11. Revised binding rules, tracking call sites of function values.

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

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

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

Fig. 12. Rules that define evaluation of previews over a dependency graph for an expression

Finally, the update function in Figure 7 also needs to be updated to store the newly created var nodes. This is done by adding the following single case:

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

8.3 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 12. 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 6), 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.

8.4 Properties of type checking

As noted in Section 8, 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 7.2, 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 7) 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 7 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.

9 DESIGN LESSONS

The motivation for the presented work, briefly outlined in Section **??**, 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 [Petricek et al. 2016]. This is not a problem in typical usage where provided members are accessed via dot. Using the worldbank example from Section 8.1, 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 6, can obtain only a delayed preview for the body of getPopulation. The delayed preview we would obtain in this case is $[\![c.\mathsf{Indicators}.\text{'Population (total)'}]\!]_c$.

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.

10 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 [Victor 2012]. Active research on novel forms of interaction happens in areas such as live coded music [Aaron and Blackwell 2013; Roberts et al. 2015]. The idea of live previews can be extended to direct manipulation [Shneiderman 1981]. The Gamma provides limited support for directly manipulating data (Section **??**), but we intend to explore this direction further.

Data science tooling. An essential tool in data science is REPL (read-eval-print-loop) [Findler et al. 2002], which is now widely available. This has been integrated with rich graphical outputs in tools such as Jupyter notebooks [Kluyver et al. 2016; Pérez and Granger 2007], 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 [Granger 2012] 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 [Burckhardt et al. 2013; McDirmid 2007]. 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 [Szwillus and Neal 1996] 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 [Perera 2013]. A promising direction is using bi-directional lambda calculus [Omar et al. 2017]. Finally, abandoning text also enables building richer, more human-centric abstractions as illustrated by Subtext [Edwards 2005]. Our current focus, however, remains on text-based editors.

Dependency analysis. Our use of dependency graphs [Kuck et al. 1981] is first-order. Building dependency graphs involving function calls using modern compiler methods [Kennedy and Allen 2001] or program slicing [Weiser 1981] would allow us to deduce possible inputs for functions and use those for previews rather than changing the language as suggested in Section 9. 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 [Consel and Danvy 1993], done in a way that allows reuse of results. This can be done implicitly or explicitly in the form of multi-stage programming [Taha and Sheard 2000]. Both can provide useful perspective for formally analysing how previews are evaluated. Semantically, the evaluation of previews can be seen as a modality [Fairtlough et al. 2001] and delayed previews are linked to contextual modal type theory [Nanevski et al. 2008], which, in turn, can be understood in terms of comonads [Gabbay and Nanevski 2013]. This provides an intriguing direction for rigorous analysis of the presented system.

11 TODO

TBD: In section about dependency graphs, include a subsection discussing how this relates to incremental computation (just like "Semantics of delayed previews" links that to CMTT etc.)

12 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

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.

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.

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.

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

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

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

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.

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.

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

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

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

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.

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.

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

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348 – 375. https://doi.org/10.1016/0022-0000(78)90014-4

Alan Mycroft, Dominic Orchard, and Tomas Petricek. 2016. Effect systems revisited—control-flow algebra and semantics. In *Semantics, Logics, and Calculi*. Springer, 1–32.

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

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.

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

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

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.

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

Peter Sestoft. 2012. *Spreadsheet technology*. Technical Report. Citeseer.

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

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.

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.

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

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

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

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

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

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

A APPENDIX

A.1 Normalization

Theorem 10 (Normalization). *For all p, there exists n and $o_1, \ldots, o_n$ such that $p \rightsquigarrow^* o_1; \ldots; o_n$.*

Proof. We define size of a program in data exploration calculus as follows:

$$\begin{array}{rcl} \mathsf{size}(c_1; \ldots; c_n) & = & 1 + \Sigma_{i=1}^{n} \mathsf{size}(c_i) \\ \mathsf{size}(\mathsf{let}\ x = t) & = & 1 + \mathsf{size}(t) \\ \mathsf{size}(e_0.m(e_1, \ldots, e_n)) & = & 1 + \Sigma_{i=0}^{n} \mathsf{size}(e_i) \\ \mathsf{size}(\lambda x \rightarrow e) & = & 1 + \mathsf{size}(e) \\ \mathsf{size}(o) = \mathsf{size}(x) & = & 1 \end{array} \tag{1}$$

The property holds because, first, both (let) and (external) decrease the size of the program and, second, a program is either fully evaluated, i.e. $o_1; \ldots; o_n$ for some n or, it can be reduced using one of the reduction rules. □

A.2 Term preview correctness

Theorem 11 (Term preview correctness). *Given a term t that has no free variables, together with a lookup table Δ obtained from any sequence of programs using* bind-prog *(Figure 6) and* update *(Figure 7), then let $v, (V, E) = \mathsf{bind\text{-}expr}_{\emptyset,\Delta}(t)$. If $v \Downarrow p$ over a graph (V, E) then $p = o$ for some value o and $t \rightsquigarrow^* o$.*

Proof. First note that, when combining recursively constructed sub-graphs, the bind-expr function 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 – the newly added nodes and edges do not introduce non-determinism to the rules given in Figure 8.

We prove a more general property showing that for any e, its binding $v, (V, E) = \mathsf{bind\text{-}expr}_{\emptyset,\Delta}(e)$ and any evaluation context C such that $C[e] \rightsquigarrow o$ for some o, one of the following holds:

a. If $FV(e) = \emptyset$ then $v \Downarrow p$ for some p and $C[p] \rightsquigarrow o$

b. If $FV(e) \neq \emptyset$ then $v \Downarrow [\![e_p]\!]_{FV(e)}$ for some e_p and $C[e_p] \rightsquigarrow o$

In the first case, p is a value, but it is not always the case that $e \rightsquigarrow^* p$, because p may be lambda function and preview evaluation may reduce sub-expression in the body of the function. Using a context C in which the value reduces to an object avoids this problem.

The proof of the theorem follows from the more general property. Using a context $C[-] = -$, the term t reduces $t \rightsquigarrow^* t' \rightsquigarrow_\epsilon o$ for some o and the preview p is a value o because $C[p] = p = o$. The proof is by induction over the binding process, which follows the structure of the expression:

(1) $\mathsf{bind\text{-}expr}_{\Gamma,\Delta}(e_0.m(e_1, \ldots, e_n))$ – 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 lookup inversion Lemma 4, $v = \mathsf{mem}(m, s)$ 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 and *compositionality* of external libraries (Definition 2), it holds that for any C such that $C[e_0.m(e_1, \ldots, e_n)] \rightsquigarrow o$ for some o then also $C[p_0.m(p_1, \ldots, p_n)] \rightsquigarrow C[p] \rightsquigarrow o$.

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 and *compositionality* of external libraries (Definition 2), it holds that for any C such that $C[e_0.m(e_1, \ldots, e_n)] \rightsquigarrow o$ for some o then also $C[e'_0.m(e'_1, \ldots, e'_n)] \rightsquigarrow o$.

(2) $\mathsf{bind\text{-}expr}_{\Gamma,\Delta}(e_0.m(e_1, \ldots, e_n))$ – This case is similar to (1), except that the fact that $v = \mathsf{mem}(m, s)$ holds by construction, rather than using Lemma 4.

(3) $\text{bind-expr}_{\Gamma,\Delta}(\lambda x \to e_b)$ – Here $e = \lambda x \to e_b$, v_b is the graph node obtained by induction for the expression e_b and $(v, v_b, \text{body}) \in E$. From lookup inversion Lemma 4, $v = \text{fun}(x, s)$ for some s. The evaluation can use one of three rules:

i. If $FV(e) = \emptyset$ then $v_b \Downarrow p_b$ for some p_b and $v \Downarrow \lambda x \to p_b$ using (fun-val). Let $e'_b = p_b$.

ii. If $FV(e_b) = \{x\}$ then $v_b \Downarrow [\![e'_b]\!]_x$ for some e'_b and $v \Downarrow \lambda x \to e'_b$ using (fun-bind).

iii. Otherwise, $v_b \Downarrow [\![e'_b]\!]_{x,\Gamma}$ for some e'_b and $v \Downarrow [\![\lambda x \to e'_b]\!]_\Gamma$ using (fun-expr).

For i.) and ii.) we show that a.) is the case; for iii.) we show that b.) is the case; that is for any C, if $C[\lambda x \to e_b] \rightsquigarrow o$ then also $C[\lambda x \to e'_b] \rightsquigarrow o$. For a given C, let $C'[-] = C[\lambda x \to -]$ and use the induction hypothesis, i.e. if $C'[e_b] \rightsquigarrow o$ for some o then also $C'[e'_b] \rightsquigarrow o$.

(4) $\text{bind-expr}_{\Gamma,\Delta}(\lambda x \to e)$ – This case is similar to (3), except that the fact that $v = \text{fun}(x, s)$ holds by construction, rather than using Lemma 4.

(5) $\text{bind-expr}_{\Gamma,\Delta}(o)$ – In this case $e = o$ and $v = \text{val}(o)$ and $\text{val}(o) \Downarrow o$ using (val) and so the case a.) trivially holds.

(6) $\text{bind-expr}_{\Gamma,\Delta}(x)$ – The initial Γ is empty, so x must have been added to Γ by case (3) or (4). Hence, $v = \text{var}(x)$, $v \Downarrow [\![x]\!]_x$ using (var) and so $e_p = e = x$ and the case b.) trivially holds.

□