# Maniposynth: Bimodal Tangible Functional Programming

Anonymous author
Anonymous affiliation

Anonymous author
Anonymous affiliation

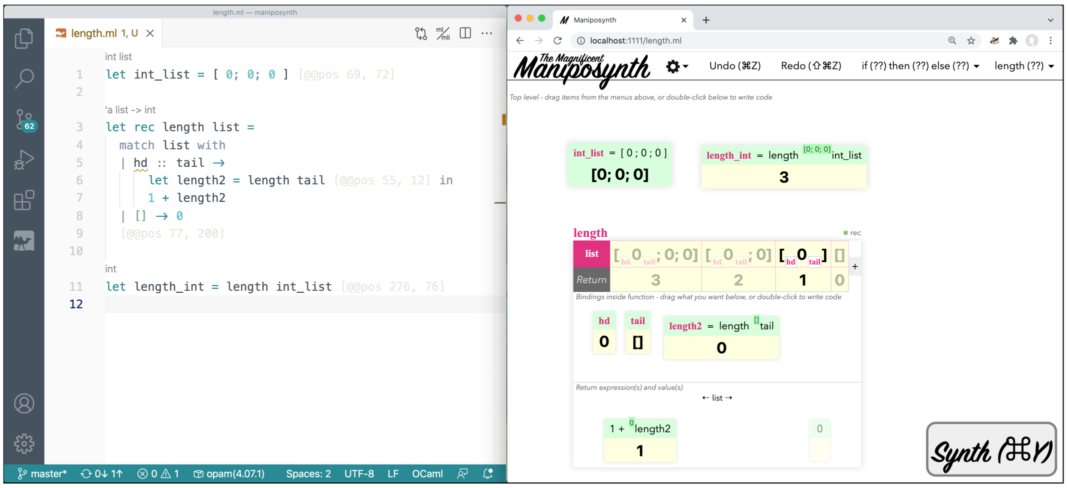


**Figure 1** A list `length` function implemented in *Maniposynth*.

## Abstract

Traditionally, writing code is a non-graphical, abstract, and linear process. Not everyone is comfortable with this way of thinking. Can programming be transformed into a graphical, concrete, non-linear activity?

While nodes-and-wires [44] and blocks-based [1] programming environments do leverage graphical direct manipulation, users perform their manipulations on abstract syntax tree elements, which are still abstract. Is it possible to be more concrete—could users instead directly manipulate live program values to create their program?

We present a system, *Maniposynth*, that re-imagines functional programming as a non-linear workflow where program expressions are spread on a 2D canvas. The live results of those expressions are displayed and available for direct manipulation. The non-linear canvas liberates users to work out-of-order, and the live values continuously show what the program is doing, and also let users interact with the live values via drag-and-drop. Incomplete programs are gracefully handled via hole expressions, which allow *Maniposynth* to offer program synthesis. Throughout the workflow, the program is valid OCaml code which the user may inspect and edit in their preferred text editor at any time.

With *Maniposynth*'s direct manipulation features, we created 38 programs drawn from a functional data structures course. We additionally hired two professional OCaml developers to implement a subset of these programs. We report on these experiences and discuss to what degree *Maniposynth* meets its goals of providing a non-linear, concrete, graphical programming workflow.

**2012 ACM Subject Classification** foo

**Keywords and phrases** Direct manipulation, tangible functional programming, programming interfaces

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23


42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:32
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

# 1 Introduction

Graphical, direct manipulation interfaces [41] are the interface paradigm most users are familiar with when they operate computers. Graphical interfaces are powerful and have extended the power of computing to almost the entire world. Nevertheless, the most powerful computer activity—programming—has proven resistant to widespread dissemination in a graphical, direct manipulation form. Most programming is primarily a text-only activity. Can general-purpose programming be reimagined in a graphical, direct manipulation interface? Experts might find productivity gains and novices might find a more approachable environment to accomplish their goals.

Most existing graphical programming approaches present the abstract syntax tree (AST) elements as items to be manipulated with the cursor. Nodes-and-wires programming environments, such as Labview [31] present program expressions as boxes whose inputs and outputs are connected by wires. Blocks-based programming environments, such as Scratch [37], present the program expressions as puzzle pieces that snap together. And structure editors, such as Barista [20], allow certain direct manipulation and transformations of the program expression as a structured entity rather than a naive string of text. What all these approaches have in common they center the interaction on the AST. This is not surprising—direct manipulation requires some concrete item to be the subject of manipulation, and the goal of programming is to produce a program, so the AST elements are a natural target for graphical interaction.

But, even more concrete than AST elements are the *values* a program produces during execution. Humans are concrete thinkers before we are abstract thinkers, and teachers know that the best way to explain is always through examples, so is there a way to write programs via direct manipulation on *values* instead of on AST elements?

The Eros environment [6] demonstrated a compelling answer to this question. Eros reimagined the programming space not as a program in text (as in traditional coding), or as a draftsman's drawing of operations connected by wires (as in nodes-and-wires programming), but as a 2D canvas of values. These *tangible values (TVs)* were primarily partially applied functions, rendered with (graphically editable!) example arguments for their unapplied inputs, with the corresponding example output displayed below (Figure 2). The user could select the output of one TV, the input of another TV (of corresponding type), and compose the two together into a new TV.

Eros highlighted that there is a complementarity between *non-linear editing* and pure functional programming. Without side effects, the order of computation is negligible. The user may gather the parts they they need, in any order they please, and worry later about how to assemble them. Alas, the standard practice of writing functional programs as linear, textual code obscures this fundamental opportunity for non-linearity. Placing values on a 2D canvas instead highlights it.

Non-linearity matters because not all humans are linear thinkers—not even all programmers are linear thinkers! A non-linear environment can offer a creative space more inviting to folks whose standard workflow naturally entails concrete exploration rather than abstract planning.

While Eros highlighted this complementarity between non-linear editing and pure functional programming, the Eros mechanism of composing TVs may tip the balance too far in favor of the concrete. Once a value has been composed, it obscures *how* it came to be. The expressions at the top of the TV (Figure 2) give some indication, but this one line is inadequate for any computation of modest size. Moreover, once composed, how does one

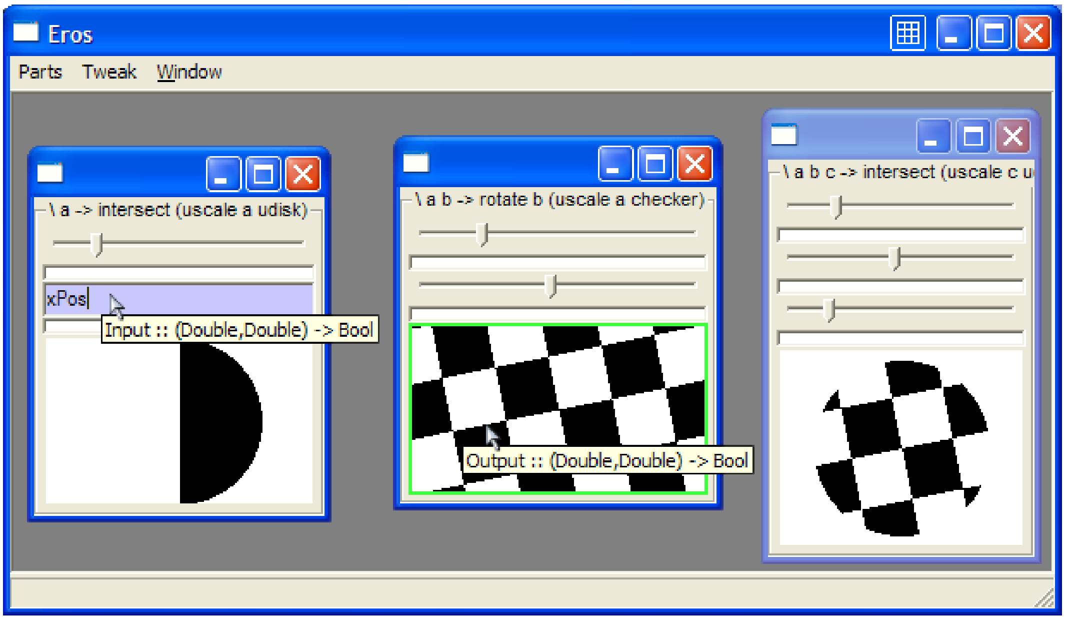


**Figure 2** (Reproduction of Figure 14 of [6]). Three tangible values on the non-linear Eros 2D canvas. Each TV shown here is a partially applied function, with unapplied arguments on top and output below. Unapplied arguments are shown with (editable) example values. For example, the leftmost TV has two unapplied arguments: a numeric argument represented as a slider (corresponding to a scale factor), and an image input (the example input image is black for positive x values and white for negative x values). The example output (the intersection of the image with a disk) is shown below. The middle TV is a function producing a checkerboard pattern, with the scale factor and rotation angle still unapplied. TVs can be composed. The user has selected the output of the middle TV and the image input argument of the left TV. Composing these together results in the rightmost TV, in which the output of the middle TV has been used as the second argument for the right TV. The remaining unapplied arguments of both are the unapplied arguments of the result TV. (Eros is a strongly typed environment, only allowing composition between outputs and inputs of compatible type. As seen in the tooltips above, images are represented as functions of type `(Double, Double) → Bool`, *i.e.* coordinates to black/white.)

change that computation that produced a TV? Value manipulation alone may be inadequate for carefully specifying abstract algorithms. Perhaps there is there a middle ground that allows both non-linear, concrete direct manipulation on values *and* traditional editing of ordinary code.

That middle ground is the subject of this paper. Here, we seek an answer to the question: **How can the approachability of non-linear direct manipulation on concrete values be melded with the time-proven flexibility of text-based coding?** We would like to create a programming environment with the following four properties:

**(a) Value-Centric.** Like Eros, and unlike most visual programming environments, we want values to be centered in the display and, as much as possible, be the subject of the user's direct manipulations.

**(b) Non-linear.** To support non-linear thinkers and exploratory programming, we allow the user to gather the parts they need out of order, and compose them together later.

**(c) Synthesis.** How to integrate recent advances in program synthesis into a practical workflow remains an open question. A value-centric interface is a natural environment to specify asserts on those displayed values, and thus also a natural environment to fulfill those asserts with a synthesizer—we want to explore this.

**(d)** **Bimodal.** Code is unavoidable: it is the language that describes computation. Ideally, a visual programming environment would not sacrifice the unique affordances of textual code—its concision and its amenability to an ecosystem of existing tooling (such as text editors, language servers, and version control). We want to offer a bimodal interface that simultaneously offers the non-linear graphical editing interface *alongside* a text-editable, traditional representation of the program's code.

## 1.1 Contributions

To show how value-centric non-linear editing can meld with traditional text-based programming, we implemented a value-centric, non-linear, bimodal programming environment with synthesis features called ***Maniposynth***. We demonstrate both how non-linear visual editing can integrate with linear code, as well as show novel editing features made possible by the value-centric display.

To gain an initial understanding of the system, we implemented an external corpus of 38 example programs.

For additional insights, we conducted an in-depth exploratory study with two external professional functional programmers, who feedback informed the evolution of ***Maniposynth***. We describe their experiences using the tool and discuss additional observations through the investigative lenses of the Cognitive Dimensions of Notation framework [11].

Section 2 introduces ***Maniposynth*** with a running example. Section 3 describes the technical implementation of the tool and the synthesizer. Section 4 presents insights from implementing a corpus of examples and the qualitative user study. Section 5 presents related work, and Section 6 discusses avenues for continued exploration.

# 2 Overview Example

To provide an overview of interacting with ***Maniposynth***, we follow a fictional programmer named Baklava as she re-implements the list `length` function from scratch.[1] Figure 1 shows the final result.

***Maniposynth*** is a locally running web application designed to be opened in a web browser alongside the user's preferred text editor. Baklava creates a blank text file named `length.ml` on her computer, starts ***Maniposynth*** in that directory, and navigates to `http://localhost:1111/length.ml` in her web browser. She positions her browser window side-by side with Visual Studio Code and is ready to begin.

## 2.1 List length, without synthesis

To start, ***Maniposynth*** displays a blank white 2D canvas. Because ***Maniposynth*** is a live programming environment, Baklava starts by creating an example list so she can see the `length` operation on concrete data. Double-clicking on the canvas opens up a text box to add new code, Baklava does so and types an open bracket `[`. Because writing example data is common in ***Maniposynth***, concrete literals up to a fixed size are offered as autocomplete

[1] Interested readers are invited to follow along in the anonymous artifact available at `http://maniposynth.org`. To configure Vim and Emacs to automatically reload changed files, jump to Appendix A. Visual Studio Code will reload changed files by default.

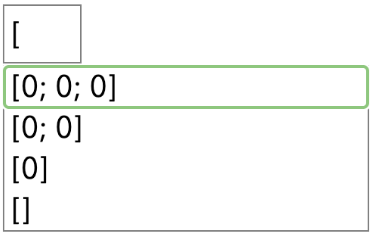


**Figure 3** List literals offered as autocomplete options.

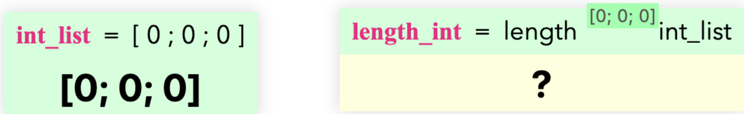


**Figure 4** Tangible values (TVs) for the example list binding and the example call to `length`.

options (auto-generated from the data constructors in scope, Figure 3). Baklava selects the list literal `[0; 0; 0]` from the autocomplete options and hits Enter.

In the code, a new let-binding for the list is inserted at the top level of `length.ml` and automatically given the name `int_list`. On the canvas, this binding is represented as a box displaying (in clockwise order, Figure 4, left) the binding pattern (`int_list`), the binding expression (`[ 0 ; 0 ; 0 ]`), and the result value below (also `[ 0; 0; 0 ]`, but bigger). These three elements together in a box form a *tangible value* in ***Maniposynth***. The box may be repositioned on the 2D canvas, and the coordinates of the position are stored in the code as an AST attribute annotation on the binding, written `[@@pos 152, 49]` in the code. Arbitrary attribute annotations are supported by the standard OCaml AST which allow these properties to be preserved across program transformations. Baklava has installed a VS Code plugin to dim these attributes in the code to avoid becoming distracted by them.

To begin work on the `length` function, Baklava now creates an example call to the function: on the canvas, she double-clicks to add new code and types `length int_list`. As before, a new binding is inserted in the code (named `length_int`) and an associated tangible value (TV) appears on the canvas (Figure 4, right). The `length_int` TV has two differences from the previous `int_list TV`. First, its result value (displayed as `?`, explained below) has a yellow background—this indicates the result is *not* a constant introduced in the expression. Second, the `int_list` variable usage in the TV's expression bears a superscript indicating the value of `int_list`, namely `[0; 0; 0]`.

In ***Maniposynth***, using a variable that has not yet defined automatically inserts a new binding (TV) for the variable—in this case, `length` was not defined. Because Baklava used `length` as a function, a new function skeleton was inserted in the code (`let length x1 = (??)`). Function TVs are displayed specially on the canvas (Figure 5). Immediately below the function name, a *function IO grid* displays the function input and output values encountered during execution. Immediately below the IO grid is a blank white area which is a *subcanvas* for the bindings (TVs) inside in the function, of which there are none yet. Below the subcanvas is a (non-movable) TV for the return expression and overall result value of the function. Currently the function return expression is a *hole expression*, written `(??)`. Hole expressions are placeholders, expected to be filled in later. For this reason, they are displayed larger than normal expressions (to make them easier targets for

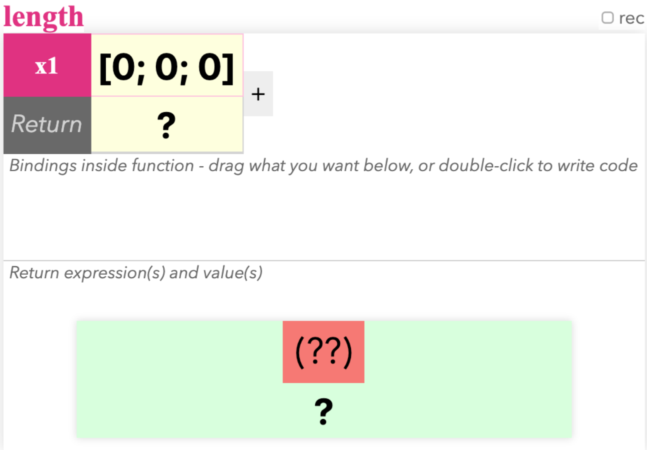


**Figure 5** Tangible value for the function skeleton binding `let length x1 = (??)`.

clicking) and have a slowly pulsing red background (to remind the user that the program is unfinished). While the `(??)` syntax is supported by OCaml's editor tooling (Merlin[2] and its language server protocol wrapper[3]), programs with holes are ordinarily not executable. To continue to provide live feedback in the presence of holes, *Maniposynth* evaluates hole expressions `(??)` to a *hole value*, displayed as `?`. This hole value `?` is the current return value of the function shown below `(??)`—in green because it was introduced by the immediate expression above—and also shown in the "Return" row of the IO grid as well as, back on the main top-level canvas, in the result value of the `length int_list` function call.

Baklava does not like the default `x1` parameter name in the `length` function and wants to rename it. Most items in *Maniposynth* can be double-clicked to perform a text edit. Baklava double-clicks the pink-background `x1` to rename the variable (patterns are pink), and writes the name `list` instead.

A goal of *Maniposynth* is to allow non-linear editing—to not need to have all of a solution before making progress. Baklava knows she must make a recursive call to the `length` function, so, without thinking hard about what might come after, she decides to add `length (??)` inside `length`. She could double-click and type this code, but typing `(??)` requires some finger gymnastics. *Maniposynth* supports a large number of drag-and-drop interactions. Any green expression can be dragged to a new location to duplicate that expression—dropping on an existing expression (e.g. a hole) replaces the existing expression, while dropping on a (sub)canvas inserts a new binding (TV). Values and patterns can also be dragged to expressions or (sub)canvases—when hovering over a value or pattern, a tooltip shows what expression will be inserted. Finally, a *toolbar* at the top of the window offers menus containing skeleton expressions: the first menu offers common expressions such as `if (??) then (??) else (??)` and `(??) && (??)` etc., the second menu offers functions defined in this file, the remaining menus offer constructors and automatically generated example values of the types in scope (the same as those offered by autocomplete)—if the user had any custom data types, they would appear here as well. Baklava drags `length (??)` from the toolbar into the subcanvas for her `length` function. A `length2 = length (??)`

---

2 `https://github.com/ocaml/merlin`
3 `https://github.com/ocaml/ocaml-lsp`

binding is created in the code and an associated TV appears inside `length`. *Maniposynth* also changes the top level `let length = ...` into `let rec length = ...`.

Because `(??)` produces hole value `?` instead of crashing, the `length` function is now diverging as `length (??)` calls `length (??)` which calls `length (??)` etc. *Maniposynth* uses fueled execution to cut off infinite loops and keep functioning. In the function IO grid, there are now extra columns showing these calls, but other than understanding why these extra columns are there, Baklava need not pay any mind that her programming is momentarily divergent.

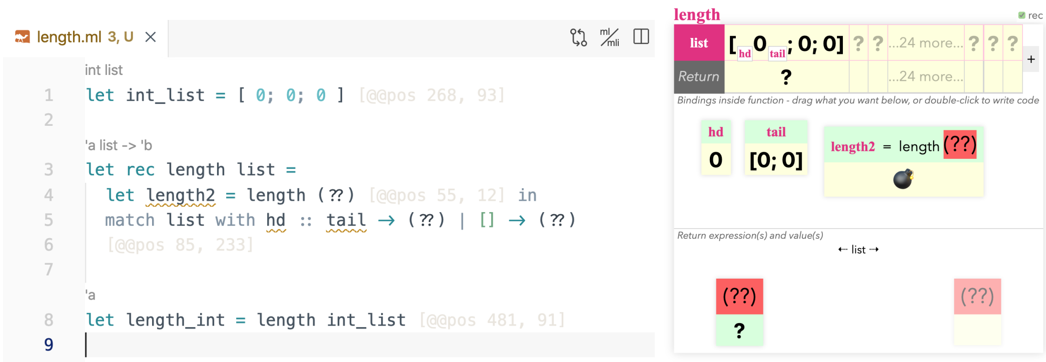


**Figure 6** Code and `length` function TV after destructing on the `[0; 0; 0]` input value.

Baklava wants the recursive call to operate on the tail of the input list. When she moves the cursor over the input list in the IO grid, a "Destruct" button appears, which she clicks. A `match` statement (*i.e.* case split) appears in her code, with holes for the return expression of each branch (Figure 6). On the display, there are a number of visual changes. In the IO grid, `hd` and `tail` pink subscripts appear inside the input list `[0; 0; 0]`, labeling the subvalues that are now bound to names by the `match` statement. To make these bindings even clearer, they are also represented as two new TVs in the function subcanvas. Finally, the function now has two possible return expressions: both appear as (non-movable) TVs at the bottom of the function, one is grayed out indicated it is not the branch taken when the input is `[0; 0; 0]`. Above the two return TVs is an indication of the scrutinee, "← list →", which allows editing of the scrutinee expression.

Now that the list tail is exposed on the subcanvas, Baklava drags it (either the pink `tail` name or the `[0;0]` value below it) onto the hole in `length (??)`, transforming it into `length tail`. In her code, the binding is moved from the top level of the function into the branch in which `tail` exists. Because *Maniposynth* embraces non-linear editing, the user should not have to worry about binding order—bindings will automatically be shuffled around as necessary to place items in the appropriate scope.

The additional calls from the recursion appear function IO grid, each still returning hole value `?`. Baklava would like to edit the base case, so she looks for the column in the IO grid where the input is `[]`, and then clicks that column to bring that *call frame* into focus. Call frames are effectively equivalent to runtime stack frames. The TVs not executed on that call are grayed out. Baklava double-clicks the no longer grayed-out return expression `(??)` and sets it to constant `0`. (She could also have double-clicked the green-background hole value `?`; values are rendered with a green background when double-clicking them will effect an edit on the expression immediately above.)

Baklava now clicks the second-to-last call frame in the IO grid to bring into focus the call where the input is `[0]`. The return expression for this branch is still `(??)`. She notes

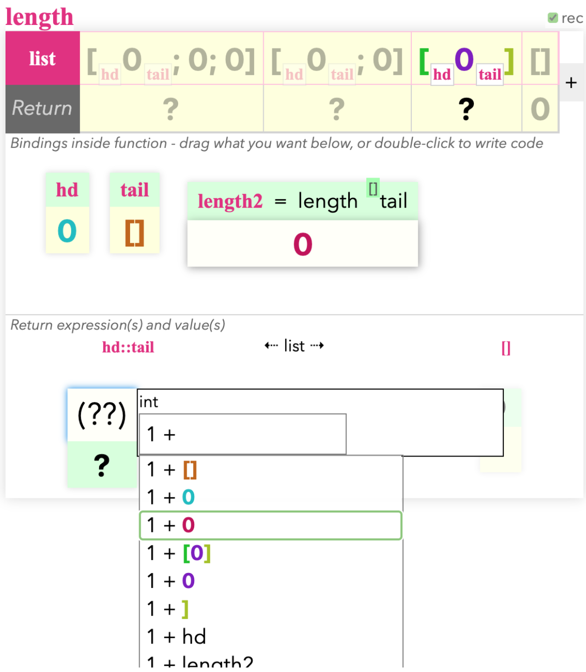


**Figure 7** Autocompleting to a value in scope.

that the TV for the `length tail` call now displays a result value of 0. Baklava double-clicks the return expression `(??)` and, after typing "`1 + `" she pauses. When she began to type, *Maniposynth* recolored the displayed values in scope in different colors, and now, looking at the autocomplete options, she sees `1 + 0`, `1 + 0`, and `1 + 0` among the possible autocompletions—each with a different color `0` corresponding to a similarly colored `0` value elsewhere on screen. The maroon `0` is the return from `length tail`, so she chooses that. The branch return expression becomes `1 + length2`, and Baklava can now see in the IO grid that her function is returned the correct value for all inputs (Figure 1).

## 2.2 Undo and delete

Baklava was an experienced programmer and did not make any mistakes. The rest of us are rarely so perfect. We should mention in passing that *Maniposynth* does support undo/redo. Additionally, any expression may be selected by a single click and then transformed to a hole by pressing the Delete key. Whole TVs (for let-bindings) can similarly be selected and deleted, removing them from the program. Uses of the binding must be deleted before deleting the binding itself—otherwise *Maniposynth* will immediately recreate a binding to satisfy the otherwise unbound variable uses.

## 2.3 Value-centric shortcuts, and synthesis

There are usually multiple ways to complete a task in *Maniposynth*. Below are a few variations Baklava might have performed instead.

**Drag-to-extract** When Baklava needed to extract the list tail and use it for the recursive call to `length`, she clicked "Destruct" on the input value and then dragged the resulting `tail` name to her `length (??)` call. The explicit “Destruct” step can be skipped. Because Maniposynth’s goal, as much as possible, is to provide manipulations on values, *subvalues* can also be manipulated. Baklava might instead have hovered her mouse over the portion of the input list `[0; 0; 0]` that is the is the tail of that list, namely `; 0; 0]`, and dragged that subvalue directly to her `length (??)` call without pressing “Destruct”. The destruction will be performed automatically and the same code will result.

**Autocomplete-to-extract** Similarly, visible subvalues are also offered as autocompletions. Perhaps the fastest way to write list `length` is, immediately after the `length` function skeleton is created, to double-click the return hole expression, type “`1 + length` ”, and then finish the new expression by selecting `; 0; 0]`, the tail of the input list, from the autocomplete options. The expression `1 + length tail` and the needed pattern match will be inserted, leaving only the base case to fill in.

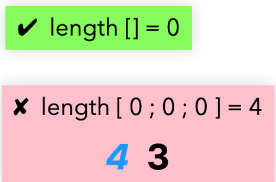


**Figure 8** A satisfied and unsatisfied assert.

**Asserts** Baklava started by creating an example call to `length`. To remind herself of the goal, she could have created an assert instead: typing `length [0; 0; 0] = 3` on the top level canvas will create an assert statement instead of a binding with a name. Asserts are rendered in red when unsatisfied, and both the expected result (in blue) and the actual result (in black) are shown. When an assert becomes satisfied, its result value is hidden and the assert turns green (Figure 8).

Asserts can also be added via the function IO grid: clicking the “+” button at the right of the IO grid will create a new column in the grid, allowing Baklava to fill in the input values and expected output. Upon hitting enter, a new assert is added at the top level.

**Program Synthesis** Asserts facilitate *programming by example*, a workflow currently available in Microsoft Excel [12] but not yet available in ordinary programming. After creating the assert `length [0; 0; 0] = 3`, Baklava might have clicked the “Synth” button on the lower-right corner of the window. Maniposynth will use a type-and-example based approach (inspired by the Myth [35] synthesizer) to guess hole fillings until the assert is satisfied or the synthesizer gives up (after between 10 and 40 seconds). The synthesizer incorporates a simple statistics model and other heuristics to improve result quality (**??**). In this scenario, with only the single assert, the Maniposynth synthesizer instantly finds a filling that creates the proper case split, but places `3` as the return of the base case and `length tail` as the return of the recursive case (Figure 9). The result is incorporated into the code, but presented to Baklava with buttons prompting her to “Accept”, “Reject”, or “Try again”. When Baklava clicks “Try again”, in about a second the synthesizer produces the correct result, which Baklava accepts.

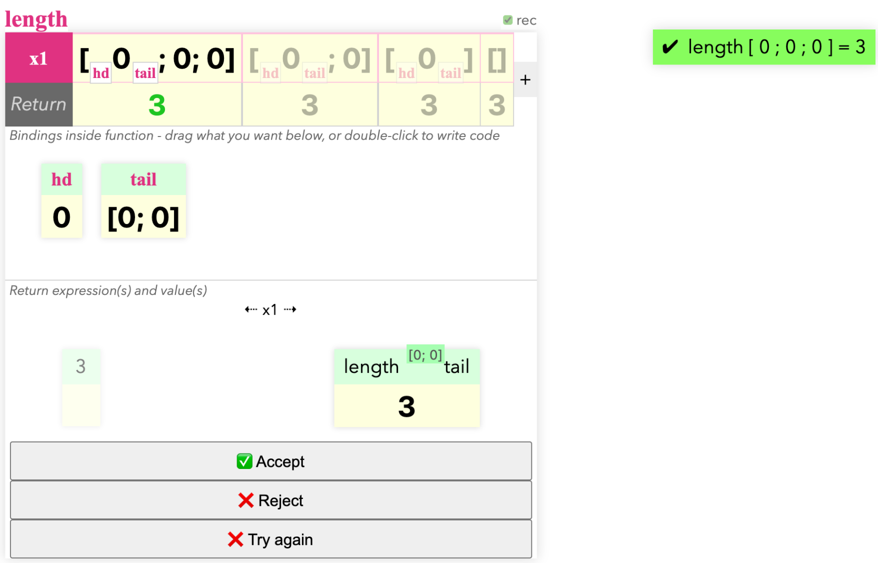

**Figure 9** An incorrect synthesis result—the next result ("Try again") will be correct.

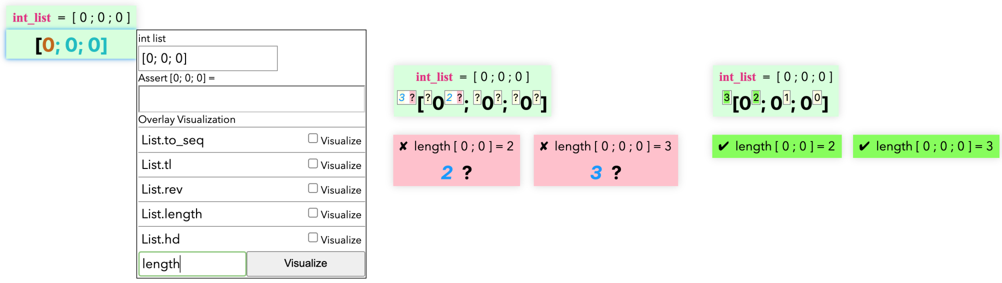

**Figure 10** Adding a subvisualization; asserts on a subvisualization, before and after satisfaction.

**Subvisualizations** *Maniposynth* offers users the ability to visualize the result of a function call on all subvalues of a displayed value—for example, the result of calling `length` on each sublist of `[0; 0; 0]` can be displayed as superscripts on the sublists (Figure 10). Asserts can also be specified on these visualized results, leading to the following workflow:

As before, Baklava first inserts [0; 0; 0] on her blank canvas. But now she clicks the [0; 0; 0] value to select it; a floating inspector window appears offering various type-compatible functions in scope to visualize atop [0; 0; 0]. Baklava foregoes these suggestions and navigates to the textbox that allows her to input a custom subvisualization. She types "length" and hits Enter. The length function skeleton is automatically created, and each subvalue of [0; 0; 0] now displays a superscript `?`, the return result of the unfinished `length` function when applied to that subvalue. Baklava double-clicks the superscript corresponding to the whole [0; 0; 0], which opens a textbox that allows her to assert on `length [0;0;0]`. She types "3", hits Enter, and the appropriate assert is created at the top level.

These subvisualizations allow users to quickly specify multiple asserts without manually creating new example values. To assert on a list of length 2, Baklava double-clicks the superscript corresponding to the tail sublist and types "2". With these two asserts, the synthesizer finds the correct result in one try.

# 3 Implementation

With the main features of the tool demonstrated, we now describe the technical operation of *Maniposynth*.

## 3.1 Architecture Overview

*Maniposynth* is a web application written in about 8600 lines of OCaml (excluding the interpreter) and 2000 lines of Javascript. *Maniposynth* relies on OCaml's provided compiler tools and AST data types to handle parsing, type-checking, type environment inspection, and pretty printing of modified code (alas, OCaml's parser discards comments). Modified code is further beautified by running it through `ocamlformat`[4] if the user has it installed.

For displaying live feedback, we need to run the program and log the runtime values flowing through the code. We modified the OCaml interpreter from the Camlboot [32] project to emit a trace of all runtime values at all execution steps. We also performed additional modifications to handle holes and asserts (described in the next section).

After *Maniposynth* runs the code via our modified Camlboot, *Maniposynth* associates runtime values from the logged execution trace with expressions in the program, and then renders HTML which is sent to the browser and displayed. Almost all OCaml-specific logic is handled server-side and baked into the HTML. The Javascript on the browser only handles TV positioning and standard GUI interaction logic. When the user performs an action, the JS sends the action to the server, the code is modified on disk, and the server prompts the browser to reload the page to re-render the display. The browser also polls the server so that when the file is changed on disk, the display will refresh.

In the next sections we describe our modifications to the Camlboot interpreter, then how *Maniposynth* handles binding reordering to provide a non-linear experience, and then the mechanics of the synthesizer,

## 3.2 Interpreter

*Maniposynth* needs to provide live runtime values. Ordinary OCaml does have a bytecode interpreter in addition to its native code compiler—ideally we could modify this bytecode interpreter to somehow emit a log of values during program execution. Unfortunately, OCaml performs type erasure and its runtime in-memory representation of values is ambiguous. It is impossible to inspect memory to recover a value's type. Remembering the types at program locations would alleviate this problem, but will fail when an expression has polymorphic type, which, alas, occurs often during program construction: the function skeleton `length x1 = (??)` has type `'a → 'b`, but in the IO grid we want to be able to display any example input values as lists, not as unknown polymorphic values. Therefore, instead of trying to modify the standard OCaml interpreter or compiler, we base *Maniposynth* off of the OCaml interpreter in the Camlboot [32] project, an experiment in bootstrapping the OCaml compiler. The Camlboot OCaml interpreter is written in OCaml and represents all runtime values as members of an ordinary OCaml algebraic data type (ADT), which allows inspecting their type and structure at runtime, at the cost of somewhat slower execution. We modified Camlboot to handle holes and asserts, and to log runtime values during execution.

[4] `https://github.com/ocaml-ppx/ocamlformat`

$$
\begin{array}{rcll}
\textbf{Programs} & p & ::= & \overline{\textbf{type}\ t\ =\ T}\ \ \overline{B} \\
\textbf{Types} & T & ::= & \text{(standard OCaml, elided)} \\
\textbf{Top-level binding groups} & B & ::= & \textbf{let}\ x = e \\
 & & | & \textbf{let rec}\ \overline{x_1 = e_1} \\
 & & | & \textbf{let}\ () = \textbf{assert}\ (e_1 = e_2) \\
\textbf{Expressions} & e & ::= & (??) \mid c \mid C \mid C\ e \mid C\ (e_1\overline{, e_i}) \\
 & & | & x \mid \textbf{fun}\ x \to e \mid e_1\ \overline{e_i} \\
 & & | & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid\ \ e_1; e_2 \\
 & & | & \textbf{let rec}\ \overline{x_1 = e_1}\ \textbf{in}\ e_b \\
 & & | & (e_1, e_2\overline{, e_i}) \\
 & & | & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
 & & | & \textbf{match}\ e_1\ \textbf{with}\ \overline{p \to\ \ e_i} \\
\textbf{Case Patterns} & p & ::= & C \mid C\ x \mid C\ (x_1\overline{, x_i})
\end{array}
$$

**Figure 11** The subset of OCaml fully supported by *Maniposynth*. Overlines denote zero or more of the syntactic element. For expressions and patterns, unsupported syntax will still be displayed but will not have full UI support.

**Supported subset** Unmodified, the Camlboot interpreter will run a large subset of OCaml. The tooling and display in *Maniposynth*, however, currently only fully supports a smaller subset, shown in Figure 11. At the top level, programs in Maniposynth are expected to consist only of type declarations followed by (potentially recursive) let-bindings; asserts are only expected to occur at the top level. Only single-name patterns have full UI support (although internal operations such as free variable analysis will account for names in nested patterns). Supported expressions include holes, base value constants, argument-less, single-argument, and multi-argument constructors, variable usages, function introductions with an unlabeled parameter, multi-argument function applications, (potentially recursive) let-bindings, tuples, if-then-else, and pattern match case splits. Case splits are only fully supported on constructors.

Records do not have complete UI support. User-defined modules, opening modules, imperative functions, and object-oriented features are currently unsupported.

The swath of supported syntax was easily enough to cover the kinds of data structure manipulations we explored in our evaluation. During the user study exercises, participants rarely missed the unsupported syntax. Even so, for the tool to become practical for everyday use, the users noted it would definitely need to support modules and imperative programming.

**Holes and Bombs** It is best for the user if live feedback is available even if the program is incomplete. While we could have the interpreter crash on the first hole, that may still be too restrictive, *e.g.* if the expression is new and is still dead code—the presence of the hole should be inconsequential to the rest of execution. A thorough solution would be to adopt the Hazelnut Live semantics, which describe how to evaluate *around* holes. [34] When holes reach elimination position, terms become stuck (*e.g.* what should hole plus hole be? Or which case branch should we take when the scrutinee is a hole?). Hazelnut Live evaluates around the term by, effectively, turning the stuck term into a value which is propagated until it causes another term to become stuck, and so on. While this can offer intriguing UI possibilities in its own right (outlined in [34]), it requires that we display the stuck terms to users as if they are values. *Maniposynth* may do so eventually, but our display is already full of elements to keep track of. Asking users to make sense of stuck terms, displayed far from their origin, might be confusing.

*Maniposynth* instead adopts a middle ground. We evaluate around holes but not around any other expressions. In practice, hole expression `(??)` introduces a hole value `?` that remembers the introduction location and captures a closure. (This closure is not displayed to the user but is occasionally used during synthesis when propagating asserts to constraints on holes.) Hole values propagate through evaluation—if unused, the evaluation can continue normally. If a hole value reaches elimination position (e.g. `? + ?`), we resolve the expression to a special Bomb value (displayed as 💣). Similarly, if a Bomb reaches elimination position, another Bomb is produced. In this way, execution can continue and expressions unrelated to the unfinished code can continue to provide live feedback.

Finally, to prevent infinite loops from stalling the interpreter or inhibiting live feedback, *Maniposynth* uses fueled execution to abort when the right-hand side of a binding takes too long to execute. Each top level let-binding is allocated 1000 units of fuel (execution steps), and each non-top level let-binding reserves 50 units for later execution in case the binding diverges. When the interpreter runs out of fuel, execution drops back to the let-binding, all patterns at the binding are bound to Bomb, and execution continues if there is any remaining fuel. Divergence is moderately common, because recursive call skeletons like `length (??)` from the Overview Example will repeated call the function with hole value. Thus it is important that execution bypasses the divergence with some fuel reserved for later bindings so the user will see later live values in the display rather having them mysteriously and suddenly disappear whenever they add an unfinished recursive call.

**Assert logging** When an assert is encountered during execution, ordinarily OCaml would evaluate the assert and then throw an exception if the asserted expression returns false. Instead, *Maniposynth* evaluates the assert and logs the result for later, but never raises an exception. The assert logging only supports equality comparisons for now. The expected expression (the right-hand side), the subject expression (the left-hand side), and the result values of both are logged. Logged asserts are used both for synthesis and to display blue expected values (Figure 8) to the user wherever that same expression and value is encountered during execution.

**Tracing** In addition to logging asserts, *Maniposynth* also logs other execution information needed to render its display. Our modified interpreter records information in two places: each execution step is entered into a global log, and we also tag side information onto runtime values.

At each execution step and at each pattern bind we add a log entry to a global trace, the log entry records the current AST location, the call frame number (from a global counter incremented upon each function call), the result value or value being pattern matched against, and the execution environment of bound variables. When producing the display, this information is queried to discover which values through which locations and under which call frame, and the appropriate values are rendered.

For convenience, we also store extra information on values as well. On values we log the type of the value when it is introduced (or returned from a built-in external primitive such as addition) so we have a concrete type associated with the value even if the value is later used in a polymorphic context. To the value we also attach a list of frame numbers and AST location of the expressions and patterns the value passes through, to, *e.g.*, conveniently interrogate where a value was first introduced. For example, to display function closure values, we find where the closure was bound to a name and display that name as the rendered closure value.

The above tracing mechanisms are sufficient to render the live feedback in the *Maniposynth* display. Although the extensive logging might be expected to slow down execution, at present *Maniposynth* is only applied to small programs and HTML rendering tends to take considerably longer than the initial execution, but the *Maniposynth* server is generally able to provide a response in under 200ms.

## 3.3 Fluid Binding Order

A primary goal of *Maniposynth* is to offer a non-linear editing experience. The program is therefore rendered on a 2D canvas, which means we do not want users to have to worry about binding ordering. If the user sees a name on the canvas, they should be able to reference that name in the expression they are editing, even if, in the written code, that name is introduced later in the program.

**Reordering bindings** To support this non-linear workflow, only limited variable shadowing is supported. Nested bindings may shadow a top-level definition, but otherwise all names are assumed to be unique within each top-level definition. After every user action, *Maniposynth* leverages these assumption to reorder bindings, move bindings into `match` statement branches, and to add a `rec` flag on bindings that refer to themselves. Only single recursion can be inferred (multiple recursion must be added manually in the text editor).

The overall consequence is that users rarely have to think about binding ordering in their code—they can continue to use the TVs on the display as if they are all appropriately visible to each other.

**Inserting case splits** Recall that users can grab any displayed subvalue and drag it into their program to induce a pattern match. Internally, the process works as follows. Whenever a user hovers their mouse over a value, a tooltip appears previewing the expression that will be inserted. For subvalues, the expression is an incomplete pattern match, such as `match list with | hd::tail -> tail`. Deeper extractions are also possible, *e.g.* `match (match list with | hd::tail -> tail) with | hd2::tail2 -> tail2`, but not often useful.

When the user drops the subvalue into their program, the expression is initially (internally) inserted as is, *e.g.* `let tail2 = match list with | hd::tail -> tail in ...`. A series of program transforms then rearranges `match` statements as follows:

1. All let-bindings at the beginning of the function are pushed down and duplicated into each pre-existing case branch. They may be pulled back out at the end of the process below. This push-down has the effect that all newly inserted `match` statements are now children of any prior `match` statements already inserted.
2. If the user dragged a subvalue that has already been extracted, (or has extracted a deeper subvalue and one of its parents has already been extracted), we do not want to insert superfluous pattern matches. We want to reuse the case splits that already exist. Relying on the prior step that ensured all bindings are now in scope of all variables introduced in cases, a static analysis pass simplifies nested case splits on the same variable: if a `match` on `list` already exists, then the copy of `let tail2 = match list with | hd::tail -> tail in ...` that was pushed into the pre-existing cons case will be simplified to `let tail2 = tail in ...`

and the copy pushed into the pre-existing empty list case will be simplified to `let tail2 = match list with in ...` *i.e.* a match with no cases, which marks the binding for removal.

3. Each let-binding that has such an empty `match` anywhere in its left-hand-side is removed.
4. Any surviving `match` statement is not redundant, but still in a non-idiomatic position. A series of local rewrite rules floats the `match` statements up to the outermost level of the function, *e.g.* `f (match list with hd::tail -> tail)` becomes `match list with hd::tail -> f tail)`, etc.
5. All let-bindings, previously floated down into all case branches, are now floated back up as far as possible to the top level of the function: a binding is pulled up outside of `match` branches when both (a) the same binding exists in all branches, *i.e.* it was valid in all branches and not removed, and (b) in all branches, the binding is not dependent (or transitively dependent) on any variables introduced for the case branch.
6. Newly inserted `match`es are now at the top level, but may still be missing cases. Incomplete pattern matches are filled in with the missing branches, with a hole expression `(??)` in each new case.
7. Bindings that are only simple renamings, such as `let tail2 = tail in ...`, are removed—these happen when the user performs an extraction of a subvalue that was already previously extracted.

The above algorithm appropriately produces idiomatic `match` statements, with the `match` wrapped in all the let-bindings that are not dependent on variables introduced in the branches. For functions with a single `match`, the above algorithm performed well in our evaluation—it was never a source of trouble. For nested matches with independent scrutinees, a current limitation of *Maniposynth* is there is no refactoring to flip the nesting order.

**Inserting undefined variables** Finally, in keeping with the goal of non-linearity, we also want to allow users to use a variable before it has been defined anywhere. Therefore, after the above processes, any remaining variables that are used but not defined are introduced in a new let-binding. Each new variable is either bound to hole or, if the variable is used as a function in an application, bound to a new function skeleton with the appropriate number of parameters. Thus, typing `length [0; 0; 0]` on an empty canvas results in the skeleton `length` function seen in the Overview Example.

## 3.4 Synthesizer

As discussed in the Overview Example, *Maniposynth* includes a programming by examples (PBE) workflow to help users finish their incomplete code. Here we detail the program synthesizer's operation and our design choices in its implementation.

The *Maniposynth* synthesizer does not contain any new "big" ideas, but the design was carefully chosen for our setting. To be as practical as possible, we had four goals:

**(a) Few examples.** To reduce the burden on the user, we would like the synthesizer to operate with few examples. For example, the Myth synthesizer [35] also targeted a subset of OCaml, but required the user-provided examples include all needed recursive calls—*e.g.* `length [0;0] = 2`, `length [0] = 1`, and `length [] = 0`. This "trace completeness" requirement is burdensome, we would like our synthesizer to operate with only one or two examples.

**(b) No type annotations.** Similarly, Myth and its successor Smyth [26] require holes to have types before synthesis, which requires manual annotation. We would like to relieve the user of this responsibility and operate without explicit type annotations.

$$
\begin{array}{llcl}
\textbf{Expressions} & e & ::= & \textbf{fun}\ x \rightarrow e \\
 & & | & \textbf{match}\ e_1\ \textbf{with}\ \overline{p \rightarrow\ e_i} \\
 & & | & c \\
 & & | & x \\
 & & | & x\ \overline{e_i} \\
 & & | & C \mid C\ e \mid C\ (e_1, \overline{e_i}) \\
 & & | & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
\textbf{Case Patterns} & p & ::= & C \mid C\ x \mid C\ (x_1, \overline{x_i})
\end{array}
$$

**Figure 12** An overview of the subset of OCaml the synthesizer can emit; also a subset of Figure 11.

**(c) As simple as possible.** The primary goal of *Maniposynth* is to explore non-linear editing, not synthesis per se, so we wanted to keep our synthesizer as simple as possible. For now, we did not adapt SMYTH because, although it appropriately relaxes the trace-completeness requirement, SMYTH utilizes a complicated synthesis schedule and requires the Hazelnut Live machinery [34] for evaluating around holes. Even with the mechanisms described below, our synthesizer is around 1300 lines of OCaml, compared to more than 5000 for SMYTH [26].

**(d) Quality results.** When given only a few examples, synthesizers are notorious for producing simple but obviously wrong results,[5] which limits their utility. This problem is compounded when the synthesizer is asked to operate in practical environments with many variables in scope, rather than unrealistic bare minimal execution environments often used for synthesizer benchmarks. Our synthesizer should operate with the OCaml standard Pervasives library open in the execution environment so the synthesizer may choose to use *e.g.* addition and subtraction. We adopt statistics and heuristics to make this tractable.

MYTH used type information to dramatically reduce the search space and to intelligently introduce case splits. So, to meet the above goals, we built a MYTH-like synthesizer which uses both types and examples to guide its guessing. Unlike MYTH, however, we relax the trace-completeness requirement and instead rely on a statistics model to guess more likely terms sooner. Our target language subset, the statistics model, and our other heuristic choices are described below.

**Synthesizable subset** Figure 12 describes the subset of OCaml that the synthesizer may produce as it attempts to fill holes in the program. The synthesizer can introduce functions, `match` statements, constants (drawn from a corpus), variable uses, function calls with a variable in function position, constructor uses, and if-then-else statements.

**Statistics model** Naively, guess-and-check will produce a large number of unlikely programs. Incorporating a statistics model to guide the synthesizer to guess more likely programs sooner and can speed up synthesis by multiple orders of magnitude [23, 18]. It also has the potential to offer the user more reasonable results when fewer examples are given.

[5] For example, "January, Febuary, Maruary" `https://techcommunity.microsoft.com/t5/excel/flash-fill-wrong-pattern-for-filling-month-names/m-p/355213`

| | | | |
|---|---|---|---|
| **Expressions** | $e$ | $::=$ | **fun** $x \rightarrow e$ |
| | | $\mid$ | **match** $e_1$ **with** $\overline{C ... \rightarrow\ e_i}$ |
| | | $\mid$ | $c$ |
| | | $\mid$ | $x$ |
| | | $\mid$ | $call$ |
| | | $\mid$ | $ctor$ |
| | | $\mid$ | $ite$ |
| **Constants** | $c$ | $::=$ | $int$ |
| | | $\mid$ | $float$ |
| | | $\mid$ | $char$ |
| | | $\mid$ | $str$ |
| **Int literals** | $int$ | $::=$ | ...integers from corpus... |
| **Float literals** | $float$ | $::=$ | ...floats from corpus... |
| **Char literals** | $char$ | $::=$ | ...chars from corpus... |
| **String literals** | $str$ | $::=$ | ...strs from corpus... |
| **Names** | $x$ | $::=$ | $local$ |
| | | $\mid$ | $pervasivesName$ |
| **Local names** | $local$ | $::=$ | $MostRecentlyIntroduced$ |
| | | $\mid$ | $2ndMostRecentlyIntroduced$ |
| | | $\mid$ | $3rdMostRecentlyIntroduced$ |
| | | $\mid$ | ...etc... |
| **Pervasives names** | $pervasivesName$ | $::=$ | ...non-imperative items in Pervasives module... |
| **Constructors** | $ctors$ | $::=$ | $localCtor$ |
| | | $\mid$ | $pervasivesCtor$ |
| **User ctors** | $localCtor$ | $::=$ | ...constructors defined in file... |
| **Pervasives ctors** | $pervasivesCtor$ | $::=$ | $e_1 :: e_2 \mid [] \mid () \mid$ false $\mid$ true $\mid$ None $\mid$ Some $e$ $\mid$ ...etc... |
| **If statements** | $ite$ | $::=$ | **if** $e_1$ **then** $e_2$ **else** $e_3$ |

**Figure 13** The grammar used for the statistics model. Each production is associated with a probability (not shown).

We model program likelihood using a probabilistic context-free grammar (PCFG). A PCFG assigns a probability to each production rule in a grammer. For our synthesizer, we derived the probabilities of the production rules from a corpus of OCaml code—namely, the source files required to build the OCaml native compiler. The overall probability of a program term is the product of the probability of the production rule of the term with (recursively) the probability of all its subterms.

Our PCFG grammar is given in Figure 13. We subdivided several kinds of terms into multiple production rules in order to provide more precise probabilities: constants are divided by the constant type, constructors are classified as either a user constructor (*i.e.* defined in the same module or a parent module) or from elsewhere, and, most notably, names are classified as local (*i.e.* defined in the same module or a parent module) or from elsewhere. User constructors are assigned equal probability with each other. Local names are denoted by how recently they were introduced into the execution environment—more recently introduced names are much more probable than less recently introduced names. The probabilities of constant literals and external (*e.g.* standard library) names and constructors are derived directly from how often those names and constructors appear in the corpus.

After calculating these probabilities from the corpus, we further reduced the search space for the synthesizer. We unscientifically trimmed down the list of possible constant literals to the 5-10 most common in the corpus for each type. We also limited the initial execution environment to constructors and functions in the globally-imported Pervasives module, removing functions involving imperative features (they are not supported by *Maniposynth*) as well as several floating point primitive operators unimplemented by Camlboot. Finally, we also excluded OCaml's polymorphic `compare`, which, in practice, the synthesizer would often use in surprising ways to produce the numbers 0 and 1, *e.g.* `compare x x` evaluates to `0`. For simplicity, we did not renormalize probabilities after the above trimmings to the production rules.

As an example, the most probable term (*i.e.* what the synthesizer should guess first) is always the most recently introduced variable. The production rule for an identifier has probability 52%, the probability that the identifier is local is 73%, and the probability a local identifier is the most recently introduced variable is 31%, for an overall probability of 12%.

**Type-based refinement** MYTH divides synthesis into two processes. The *type-based refinement* process introduces program sketches—either function introductions or case splits—at holes based on the type at the hole and the types of variables in scope (to find an appropriate scrutinee to introduce a `match`). These sketches contain further holes to fill (*i.e.* for the function body and the match branches). Type-based refinement alternates with the *type-directed guessing* process, which performs simple type-constrained term enumeration to guess a term to fill existing holes (guessing will not introduce functions or `match` statements).

As part of the type-based refinement process, MYTH will push the user's examples to the frontier of synthesis. For example, if the user asserts that a hole should output `0` when given the input `[]`, MYTH will refine the hole to `fun x -> (??)` and refine the example to note that `(??)` should resolve to `0` when `x` is bound to `[]`. This allows MYTH to quickly verify when a hole filling satisfies all given examples.

However, MYTH's implementation of this example refinement machinery requires that the user provide all asserts directly on holes. Users cannot write `assert (length [] = 0)`. Instead, they must write, essentially, `let length = ((??) such that { [] => 0 })`. It would be better if users could invoke synthesis on a partial sketch.

To allows asserts on program sketches rather than only on holes, SMYTH uses "Live Unevaluation" [26] to push top-level asserts down to constraints directly on holes. Pushing down the asserts is not fundamentally required to perform synthesis—a synthesizer may guess terms at the holes and check the top-level asserts (indeed *Maniposynth* does so)—but pushing the asserts down to the holes provides information about the hole. *Maniposynth* adapts an effectively[6] identical approach to SMYTH, and refines the examples through the sketch yielding constraints on holes. *Maniposynth* uses these hole constraints for two purposes:

[6] Some sketches prevent the propagation of constraints—for example, if a sketch has a hole in scrutinee position, it is impossible to know which branch to take. The push-down procedure is stuck and any holes in the branches will not be able to receive their constraints. SMYTH resolves this scenario by speculatively filling the scrutinee hole while pushing down the constraints. We opt to avoid this extra machinery, resulting in a simpler implementation, but at the cost that we cannot resolve holes in an iterative fashion—satisfying the requirements of one hole before moving on to the next—which would enable faster synthesis.

1. Refining a hole into a function requires knowing that the hole is at arrow type. This can easily be determined if the program is explicitly typed, as required in Myth and Smyth. ***Maniposynth***, however, allows untyped sketches which often start at polymorphic type. For example, in an initial sketch `let length = (??)`, the `length` variable has type `'a`. When `assert ([] = 0)` is pushed down to the hole, we know the hole must satisfy the requirement $[] \Rightarrow 0$, *i.e.* must be a function that when given `[]` produces `0`. If all the constraints on a hole are of that form $v_1 \Rightarrow v_2$, then ***Maniposynth*** will attempt to refine the hole into `fun x -> (??)`.
2. To speed synthesis and produce more relevant results, ***Maniposynth*** tracks whether a generated (sub)term is allowed to be constant or not (*e.g.* ***Maniposynth*** requires that at least one argument in a function must be non-constant). If multiple different examples reach a hole, ***Maniposynth*** will not generate a constant term for that hole. Similarly, if a single example reaches a hole, ***Maniposynth*** will exclude all constants from consideration for that hole, except for the value asserted on the hole.

***Maniposynth*** currently only performs at most one level of refinement—introducing only one function or one case split. Further (or initial) case splits can be introduced by the user with the "Destruct" button in the UI. Introducing functions is rarely needed in practice because, in the Maniposynth UI, undefined variables are inserted with a function skeleton.

**Type-directed guessing** Terms are enumerated (guessed) at holes up to a given probability [22]. During term enumeration, the probability bound is treated as a resource that is iteratively distributed between holes, and then between guessed subterms. When the probability is exhausted, no further enumeration occurs on a subtree. If a candidate subterm's probability is above the final probability bound, the remaining probability is available for enumerating sibling terms.

Within a hole, term enumeration is type-directed, starting from the type of the hole. Leveraging OCaml's type checking machinery, subterms are unified during the enumeration process to narrow the type. For example, if a hole has type `int` and the synthesizer guesses a call to `max` which has type `'a → 'a → 'a`, the return type will be unified with `int` and the synthesizer will only guess terms of type `int` for the arguments.

As discussed above, initial sketches often have polymorphic types unhelpful to the synthesizer. To tighten these bounds before term enumeration, the input and output types of functions are speculatively chosen based the given examples. If the given examples differ in type, multiple speculative types are explored (terms are guessed in each). Future versions of ***Maniposynth*** may use anti-unification instead to be more precise. The speculative types are not included in the final synthesis result in case the inferred code has a more general type.

As briefly mentioned, ***Maniposynth*** limits where constant terms may appear, in order to reduce the number of unnatural synthesis results. The term enumeration eagerly tracks whether a term may be constant or not. A term is estimated to be non-constant if it uses any introduced function parameter, or any variable introduced under the outermost enclosing function. At most one hole may be constant, and, when introducing a function call, at least one argument must be non-constant. If a previously filled hole is constant, or we have reached the last function argument and all other arguments are constant, then the subterm enumeration will avoid enumerating constants. This speeds synthesis and produces more reasonable results.

**Final heuristics** Finally, when all holes have been filled with type-appropriate terms within the probability bound, the candidate program is accepted if:

**(a)** All asserts are satisfied. (Fueled execution prevents divergence when checking asserts.)
**(b)** At most one hole is filled with a constant.
**(c)** All introduced function parameters are used.
**(d)** The result at a hole has not previously been rejected by the user.

If no satisfying hole fillings are found at the initial probability bound and a 10 second timeout has not been reached, guessing is restarted with a new bound 1/20 of the old. If there is a valid candidate program, the highest probability such program is returned. Enumeration within a given probability bound is not precisely from highest to lowest probability, however, so timeout will not interrupt a round of synthesis until the full space of that probability bound is explored. Thus for the user, the timeout they experience varies between 10 and 40 seconds.

# 4 Evaluation

To evaluate to what degree *Maniposynth* meets its goal of providing value-centric, non-linear editing, we performed two evaluations. In one, an expert user (the first author) used *Maniposynth* to implement 38 functions from the exercises and homework of a course on functional data structures [33]. In the second, to provide additional qualitative insights on the operation of the tool, we hired two professional OCaml programmers, guided and observed them as they used *Maniposynth* to implement a subset of the exercises from the above course.

## 4.1 Study Setups

The first six lessons of the course [33] cover natural numbers (via an ADT), various list functions, leaf trees, binary trees, binary search trees, and a form of binary search tree that also records on each node the minimum value of all its descendants. We excluded the six functions on this specialized tree because of time constraints. The course exercises and homework spanned 38 functions on the remaining data structures. The first author implemented each of these functions in *Maniposynth* with the code editor hidden. *Maniposynth* was set up to log the number and kinds of actions that were preformed. We report on these in the next section.

For our user study, we hired two professional OCaml programmers to use the tool for three sessions each, spread over three weeks, with each session lasting two hours. Participant 1 (P1) and Participant 2 (P2) had 5 and 11 years, respectively, of professional OCaml experience. The participants ran *Maniposynth* on their own computers alongside their preferred text editor (Vim in for both). The study facilitator connected via video conference and recorded the sessions. Participants implemented their choice of exercises from the list, or suggested their own task to complete. The facilitator provided varying amounts of guidance throughout, starting with close guidance to teach the tool and transitioning to less intervention as participants became more comfortable. After each exercise and at the end of each session, participants discussed a series of questions posed by the facilitator. In concert with *Maniposynth*'s four design principles—value-centric operation, non-linearity, supporting synthesis, and bimodality—we aimed to gain insights about the following four research questions, along with three supplemental questions:

| Function | LOC | Asserts | Time | Mouse | Keybd | Un/Re/Del | TypeErr | Crash |
|---|---|---|---|---|---|---|---|---|
| nat_plus | 5 | | 0.8 | 6 | 5 | | | |
| nat_minus | 8 | | 1.9 | 6 | 11 | | | |
| nat_mult | 9 | | 1.4 | 8 | 6 | | | |
| nat_exp | 13 | | 2.1 | 9 | 6 | | | |
| nat_factorial | 13 | | 1.6 | 8 | 4 | | | |
| nat_map_sumi | 10 | | 2.6 | 11 | 5 | | 1 | |
| count | 9 | | 1.9 | 9 | 11 | | | |
| length | 4 | | 0.3 | 1 | 7 | | | |
| snoc | 8 | 1 | 2.4 | 8 | 12 | 2 | | |
| reverse | 8 | | 1.5 | 4 | 9 | | | |
| nat_list_max | 17 | | 4.6 | 23 | 21 | | | |
| nat_list_sum | 13 | | 1.1 | 9 | 4 | | | |
| fold | 9 | | 3.2 | 14 | 6 | | | |
| shuffles | 14 | | 14.5 | 25 | 28 | 2 | | |
| contains | 9 | | 2.2 | 10 | 13 | 1 | | |
| distinct | 16 | | 2.4 | 9 | 11 | 2 | | |
| foldl | 10 | 1 | 1.5 | 10 | 6 | | 1 | |
| foldr | 8 | 1 | 1.8 | 10 | 5 | | | |
| slice | 12 | 3 | 9.8 | 19 | 22 | 4 | | |
| append | 8 | 1 | 1.4 | 7 | 9 | | | |
| sort_by | 21 | 3 | 6.2 | 17 | 29 | | | |
| quickselect | 13 | 1 | 13.1 | 19 | 38 | 1 | 1 | |
| sort | 16 | 3 | 5.6 | 11 | 32 | 2 | | |
| ltree_inorder | 12 | 1 | 2.9 | 7 | 20 | 1 | 1 | |
| ltree_fold | 13 | 1 | 3.1 | 13 | 13 | | | |
| ltree_mirror | 11 | 1 | 4.4 | 12 | 6 | | 1 | 1 |
| bst_contains | 14 | 3 | 6.6 | 11 | 32 | 1 | | |
| bst_contains2 | 17 | 5 | 10.4 | 20 | 41 | 2 | | |
| btree_join | 34 | 2 | 61.7 | 82 | 64 | 51 | | 2 |
| bst_delete | 36 | 2 | 14.4 | 31 | 24 | 4 | | |
| bstd_valid | 29 | 3 | 32.2 | 63 | 100 | 4 | 1 | |
| bstd_insert | 18 | 2 | 8.0 | 38 | 23 | 3 | | |
| bstd_count | 21 | 1 | 7.6 | 15 | 32 | 1 | | |
| bst_in_range | 31 | 3 | 9.3 | 23 | 39 | 3 | | |
| btree_enum | 29 | 3 | 19.2 | 31 | 51 | 6 | 3 | |
| btree_height | 15 | 1 | 1.9 | 11 | 14 | | | |
| btree_pretty | 14 | 1 | 3.7 | 4 | 21 | | 4 | |
| btree_same_shape | 19 | 1 | 8.1 | 14 | 34 | 7 | | |
| **Total** | **566** | **44** | **277.6** | **628** | **814** | **97** | **13** | **3** |

**Table 1** Example exercises, with lines of code, number of asserts, time in minutes, number of mouse actions (excluding selection and undo/redo), number of keyboard interactions (*e.g.* typing in a textbox), number of undo/redo/deletions, number of type errors encountered, and number of times *Maniposynth* crashed and the file had to be repaired in the text editor.

**RQ1.** How do users interact with the live values?
**RQ2.** How do users work non-linearly?
**RQ3.** How do users interact with program synthesis?
**RQ4.** How do users interact with their text editor?
**SQ1.** What are the pain points? How might the system be improved?
**SQ2.** How comfortable are participants with the tool—can they complete an exercise without guidance?
**SQ3.** Using the lenses of the Cognitive Dimensions of Notations [11], what additional insights do we learn about the tool?

For each participant, the first session introduced to the tool without synthesis, the second session introduced synthesis (before the synthesizer had a statistics model), and the third session concluded with the tool as presented here. Based on participant feedback, we fixed bugs and made improvements between each session.

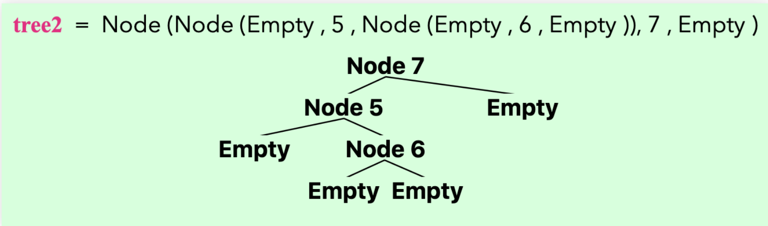


**Figure 14** *Maniposynth* beautifies tree-like values.

## 4.2 Results

**Example corpus implementation** The expert implementer took about 4.5 hours to implement the 38 functions, resulting in about 550 lines of code (including AST annotations and examples written to have live feedback, but excluding whitespace). A quantitative summary of these example exercises is shown in Table 1. There are often many paths to a correct implementation, so to provide some constraint the implementer did not use synthesis, and did not use the ordinary text editor except to copy an earlier function into a later exercise (in case of dependencies) or when *Maniposynth* crashed on the given code. For the functions operating on tree-like datatypes (ADTs with multiple children of the same recursive type), *Maniposynth*'s live display helpfully draws the trees as trees (Figure 14). Primarily, these 38 examples show that the *Maniposynth* UI is expressive enough to create these programs. We also noticed two qualitative takeaways from the exercises. First, although we believe bimodality is an important property for the grounding and long-term practicality of the tool, it is possible to hide the text editor and work entire in *Maniposynth*. Second, even with live feedback available, it is not always used—a theme that reemerged in our user study. We now discuss these two observations.

The implementer used *Maniposynth* in fullscreen with their text editor hidden. The text editor was only used to initialize code by pasting from a previous exercise if prior code was needed, and in cases where *Maniposynth* could not run the code and crashed (*e.g.* when the implementer tried to raise an exception in unreachable code, but instead raised the exception in reachable code!). Overall, *Maniposynth* operated well, even without the textual view, with a couple notable caveats.

The non-linearity machinery largely worked—the implementer did not have trouble with binding order. Even so, the implementer was careful to name extracted subvalues well, because the positioning of the extracted TVs on the 2D display did not (by default) reflect the items' positions in the original data structure. Particularly for nested matches, there were sometimes a large number of these extracted values displayed and it was hard to keep track of them. At least once the implementer repositioned the extracted TVs (the TVs representing case split branch patterns) to reflect those original positions. Ordinary textual code for case split patterns would provide some of these positional cues with manual interaction.

On a few exercises where nested `match` statements were needed, *Maniposynth* initially created the wrong nested `match` structure; with the non-linear display, this is a bit hard to notice and requires thinking about the `match` nesting structure shown in the return TVs area. Regardless, the implementer was able to work around the trouble by undoing and triggering the destructions differently.

The second observation from these examples is that, when working with *Maniposynth*, the implementer noticed that they seem to flip between two mental modes: these correspond roughly to focusing on displayed values vs focusing on expressions. In the *value-oriented mode*, the implementer would put their attention on the live values to consider if the code is operating correctly; in the *expression-oriented* mode, the implementer would read expressions and simulate the computer's operation in their head. As a matter of discipline, the implementer was trying to push themselves to consider and use the live values, but still often found themselves reverting to thinking only about the expressions instead. We have three hypotheses for why there seems to be a tendency to revert to focusing on expressions instead of values. Hypothesis A: Expressions are a concise language that represent abstractions, and programming is, fundamentally, abstract. Language is how humans handle abstraction and so our brains are good at language. The concrete values do not immediately represent the abstraction. Hypothesis B: Seasoned programmers have years and years of experience reading code and simulating the computer in their head, our brains have adapted to it and it feels natural. Hypothesis C: ***Maniposynth*** did not provide enough live feedback and forced the implementer to consider the expressions. In some cases this was immediately true: ***Maniposynth*** currently only displays the first and last three call frames, with no option to see the others. Despite the implementer's resolve to try to work with values, sometimes those values were in unavailable call frames. Additionally, when initially trying to figure out what algorithm was needed at all, the implementer found it easier to work out the initial sketch in their head rather than guess and check in ***Maniposynth***. Most likely, all three of these reasons contributed towards a tendency to put attention back on expressions rather than values. A similar theme was observed in the user study.

**RQ1. How do users interact with the live values?** This theme of value-oriented focus versus the "old way" of expression-oriented focus appeared in several of the participant's interactions with the tool. For example, despite the values featuring prominently in the display, it took until after the entire first exercise for P2 to fully realize they were looking at and working with live *values*. In another scenario, P1 and the facilitator together spent an embarrassingly long time trying to find a bug into an `insert_into_sorted_list` helper. After finding the bug they realized that, had they inspected the live values more closely, they might have found the bug much sooner. Additionally, P2 observed that they are so used to reading trees as long lines of serialized text (*e.g.* `Node (Leaf 2, Node (Leaf 2, Leaf 3))`) that they were subtly repelled by the beautified 2D rendering of tree values: "I see the splayed-out tree [rendered tree] and I'm like, 'Oof, I can't read this,' even though it's much more readable!"

Even so, participants still did use the live display and expressed appreciation for it. For example, P2 also noted that, when working with trees in ordinary programming, if their function didn't work they would be forced to write large amounts of tree pretty-printing code to perform printf-debugging; the live display ameliorated that issue. (And P2 wished we had introduced trees earlier in the study so they would have had more time to play with them!)

Live values require the user to switch call frames to see other example function calls, or calls that hit a different branch in the code. This was not always natural for participants. In the first session, P2 admitted to sometimes being confused about what branch they were looking at. And, despite gaining moderate proficiency with the tool by the end of the study, P2 still remarked that it was hard to think about how you can flip between frames. How to modify the display to help clarify this operation remains an open question.

**RQ2. How do users work non-linearly?** We wanted to know how participants adapted to ***Maniposynth***'s non-linear style. The tool requires a number of "inside-out" (P1) changes in thought, such as creating an example before defining a function, providing expressions *without* naming them first, and not worrying about let-binding order but instead just using an out-of-scope variable and letting *Maniposynth* move the binding. By the end of the study, participants were familiar with these concepts but did necessarily start out that way. For example, in the first session P1 had trouble remembering to create functions by first providing an example call, but by the end of the study was doing so without any prompting from the facilitator. P1 also initially had trouble finding items on the screen by name but felt more comfortable by the second session. Near the end of the second session P2 expressed, "I want a let binding. . . I don't have any confidence I can make let bindings," despite having successfully done so many times by double-clicking the subcanvas or dragging values into the subcanvas. P2 instantly understood after a quick reminder from the facilitator, but it is notable that even after around three hours with the tool it hadn't quite sunk in that most TVs are let-bindings.

At the end of the study, we asked the participants their thoughts about writing expressions without naming them first. P1 expressed they would more likely prefer to instead always have to provide a name; P2 was unsure, but noted that *Maniposynth*'s default names had improved from the first version we had them try. In particular, at P2's behest we hard-coded the default names for list destruction to be `hd::tail` instead of the original type-based `a2::a_list2`. Even so, we rediscovered that naming was important in programming. Function skeletons are still inserted with generic parameter names, *e.g.* `fun x2 x1 -> (??)`, which are both unhelpful and backwards. This indeed resulted in user mistakes during the study, and is a point to improve in future versions of *Maniposynth*.

Despite a few troubles, both participants were positive overall about the non-linear workflow. P1 noted the non-linear style "fits a lot more with how I like to write code," and P2 said, "I like it, I'm excited about it."

**RQ3. How do users interact with program synthesis?** We introduced participants to the synthesizer in the second session, at which point the synthesizer lacked a statistics model (instead enumerating terms from small to large) and did not offer the "Accept / Reject / Try again" buttons (instead requiring the user to Undo on an incorrect synthesis result); these were added for the final session. We wanted to know how comfortable users were with providing asserts and using the synthesizer.

Participants were familiar with thinking in asserts. In the first session, the facilitator only introduced participants to providing example function calls, not asserting on their results. Despite this, unprompted, both participants wanted to make asserts once they had an example to work with. When asserts were formally introduced, participants were generally comfortable providing examples, although P1 would occasionally write asserts in a polymorphic form, *e.g.* `foldl f acc [] = acc`, which would insert new blank bindings for `f` and `acc` on the canvas and P1 would have to recover from the mistake. Even so, P1 appreciated that *Maniposynth* encouraged them to write in a test-driven development (TDD) style, and suspected it prevented them from making simple errors. When asked if they had trouble writing assertions, P2 responded, "I had trouble *not* making assertions," because P2 enjoyed toying with the synthesizer, but P2 did observe that constructing trees was a little tricky. *Maniposynth* only renders beautifies tree values, not tree literal expressions. In the future, *Maniposynth* may beautify tree expressions in addition to values. Overall, we asked participants to rate how laborious it was to create examples, P1 and P2 responded with 2 and 4, respectively. Providing asserts was not a bottleneck.

The facilitator introduced synthesis to the participants with the list `length` example, which left a positive first impression on the participants. Synthesis was somewhat less helpful after the `length` example. By the third session, the synthesizer was usually able to finish participants' functions that were mostly sketched-out (if sketched out correctly!), but occasionally still failed. Even so, participants appreciated the synthesizer when it succeeded and were not bothered when it did not.

A prior study of synthesizer users revealed that users will sometimes accept synthesis results they do not understand, but in that study it did not lead to correctness errors [8]. Our study does provide one counterexample: when P2 invoked synthesis to fill out the final else-branch for BST insert, P2 examined the resulting expression and did not notice that it erroneously duplicated the right subtree; had they written the expression by hand, it is possible they would not have encountered this mistake. It would have been helpful if the synthesizer highlighted the ambiguity at that location, perhaps by finding solutions within a certain probability bound of the most likely solution and leveraging an interface similar to [28] to allow the user to choose between alternative subexpressions.

When initially introduce to the participants, the synthesizer did not have the "Accept / Reject / Try again" buttons, instead participants were required to Undo, but often forgot to do so. Without those buttons, there was also no feedback in ***Maniposynth*** that clearly indicated what had changed—P1 admitted to looking at their Vim window to ascertain what the synthesizer produced. The addition of the "Accept / Reject / Try again" interface was appreciated by participants and P1 noted this did keep their focus more on the Maniposynth window.

Overall, the facilitator's impression was that the participants were comfortable trying to use synthesis, but did not necessarily obtain mastery of it, in part because synthesis is opaque. P1 noted, "It is really hard to know whether synthesis is failing because I have posed the problem in an incorrect way or synthesis is failing because I haven't given it a lot of information. But the process of trying to give it more information is very illuminating in terms of whether my conception of the problem is wrong." P2 as well initially felt that working with the synthesizer was unfamiliar but remained intrigued by its potential, saying, "It was kind of awkward at first. It sort of seemed like a cool trick but there were parts where it would actually complete the program which was kind of nice even though it was not like a very trivial program. That's a neat feature." These experiences suggest that synthesis in this setting is a viable workflow, despite its initial unfamiliarity.

**RQ4. How do users interact with their text editor?** Participants were not forbidden from using their text editor, but the heavy focus on learning Maniposynth meant that they only did so only as a last resort. When asked, P1 estimated they spent about 40% of their time looking at Vim when they were trying to figure out what was going on, but, by the end of the third session, only felt the need to edit in Vim on particularly tricky errors. P2 also felt more comfortable in Vim, "When I was really stuck, I felt self-conscious and I was like, 'Alright I'll just figure this out in Vim quickly.' It's faster, probably, I've got years of experience doing that."

Part of the promise of bimodal editing is that one *can* do this! Even so, participants performed the vast majority of their editing the in the ***Maniposynth*** display. As noted, the participants only occasionally needed to edit in Vim, but even when operating ***Maniposynth*** they did seem to rely on looking at the textual display to understand what was happening. As noted below, ***Maniposynth*** may be over-reliant on shapes and colors to differentiate different kinds of elements and it can be confusing, which may have driven the participants to look at their Vim window instead of relying solely on the ***Maniposynth*** display.

**SQ1. What are the pain points? How might the system be improved?** The participants had trouble keeping track of what everything was in the *Maniposynth* display. Maniposynth relies on colors and shapes to distinguish what different UI elements are: expressions, values, function parameters, asserts, expected values, return expressions, patterns, let-bindings (TVs), and different (sub)canvases that hold let-bindings. Both participants expressed a desire for more explicit labels of what all these different elements were. After the first session, we added labels on the various subcanvases ("Top level", "Bindings inside function", "Return expression(s) and value(s)") which P1 expressed appreciation for. We had hoped those would obviate the need for more labeling, but even by the end of the final session the participants still expressed a desire for more indication of what different elements were.

**SQ2. How comfortable are participants with the tool—can they complete an exercise without guidance?** After each exercise we asked participants if they felt comfortable completing the next task without assistance from the facilitator. By the end of the final session P2 was comfortable with minimal assistance, whereas P1 still felt the need for help—although this impression is somewhat confounded because the later exercises participants attempted were more challenging.

**SQ3. Using the lenses of the Cognitive Dimensions of Notations, what additional insights do learn about the tool?** The Cognitive Dimensions of Notations [11] is a framework of twelve lenses for qualitatively assessing design trade-offs. Below, we report a subset of our observations from considering these lenses.

*Diffuseness (How noisy is the display?)* *Maniposynth* stores extra information, such as 2D binding coordinates and previously rejected synthesized expressions, as annotations in the OCaml code. P1 opined that, "All the annotations do make it less attractive to try to do stuff in Vim," and theses annotations were a source of confusion. The rejected synthesized expressions were particularly confusing because the whole discarded expression was in text in the code, albeit wrapped with `[@not ... ]`, and participants would sometimes read these large expressions without realizing it was not the code they cared about. After the user study, we modified *Maniposynth* to store a short hash of the rejected expression rather than a full copy. Additionally, *Maniposynth* includes a syntax highlighting rule that will gray out AST annotations, but it only works in VS Code with the Highlight extension installed [43].

*Secondary Notation (Is there non-semantic notation to convey extra meaning?)* *Maniposynth* does not currently support comments. P1 missed having comments, while P2 did not.

*Viscosity (How hard is it to make changes?)* Three main scenarios arose where changes where difficult. First, editing a base requires that some execution hits the base case, otherwise the base case can never be focused; this was occasionally a hindrance and might be addressed either by adding a "phantom call frame" that focuses the case without a concrete execution or by automatically synthesizing an example that hits the case. Second, once an expression was in the program, it was hard to wrap the existing expression with some new expression; it would be better if there were a mechanism to indicate whether a new drag-and-dropped expression should replace or wrap the old. Finally, (sub)expressions could be text-edited by double-clicking them on the display. However, sometimes participants (and even the first author) would double-click a subexpression but instead want to edit a parent of that expression, which required a different interaction. Future versions of *Maniposynth* may, upon double-click, open the entire expression for editing but with the clicked subexpression initially selected out of the whole line of code.

*Visibility (Is everything needed visible? Can items be juxtaposed?)* Element positioning in *Maniposynth* proved tricky, because elements will change size based on the size of the values in the TVs—multiple large trees in the function IO grid, for example, can make a function take up the whole window. Participants did have to move asserts around. P2 used a large screen and expected their functions to grow rightward: P2 would position asserts far to the right of their nascent function. P2 also expressed the desire for a snap-to-grid so they could align their TVs perfectly. P1 used a smaller screen which may have caused trouble: at one point P1 was trying to debug and realized after-the-fact that they had scrolled the IO grid offscreen—had it been onscreen and they looked at it, they might have found their mistake quicker. One possible mitigation is to scale down large values.

## 5 Related Work

A number of systems share *Maniposynth*'s goal of centering the programming workflow around live program values.

**Programming by demonstration (PBD)** Programming by demonstration (PBD) is an interaction paradigm in which the user demonstrates an algorithm to the computer step-by-step, resulting in a program. The first PBD system, Pygmalion [42], targeted generic programming and, like *Maniposynth*, displayed the live values in scope as the subject of the user's manipulations. For example, a function call with missing arguments was represented as an icon on the canvas. When all arguments to a function call were supplied, the icon for the function call was replaced with a display of its result value. To use that result value, the user dragged the value to where they wanted to use it. Recursion was supported. Although the 2D canvas was non-linear, Pygmalion treated the program as an imperative, step-by-step movie over time and did not offer a corresponding always-editable text representation.

Like Pygmalion, Pictorial Transformations (PT) [16] also offered program construction via step-by-step manipulation of a display of the live program values. PT allowed the user to custom visualizations, and was in general more expressive than Pygmalion, supporting more complicated algorithms, including those involving lists. Later PBD systems were usually more domain-specific [3, 25], although ALVIS Live [17] targeted the construction of iterative array algorithms by demonstration, and, notably, represented the resulting program in ordinary text.

Some empirical evidence for the possible benefits of a value-centric workflow was provided by the Pursuit PBD system for creating shell scripts [30]. In evaluating Pursuit, it was discovered that a comic-strip style representation of a program—with before and after values represented in the frames of the comic-strip—enabled users to more accurately generate programs compared to a more textual representation.

**Live nodes-and-wires** In 2D nodes-and-wires programming [44], nodes usually represent transformations (expressions) and the wires represent dataflow (values). Consequently, nodes-and-wires environments do not necessarily display live values, although some do output live values below the nodes (*e.g.* natto.dev [40]). Among these environments, Enso [7] is also bimodal, like *Maniposynth*, offering both textual and graphical representations for editing the program.

PANE [15] flips the usual node-and-wires paradigm and instead uses example values for the nodes and locates transformations (expressions) on the wires, placing values even more at the center of attention compared to its peers. Examples values can even be clicked to invoke operations on them. PANE does not, however, maintain an editable text representation of the program.

**Live programming** Like *Maniposynth*, traditional live programming research seeks to augment ordinary, text-based coding with display of live program values, although the displayed values are read-only. There are a growing number of such systems. Python Tutor [13] is popular teaching tool for visualizing Python program state. Bret Victor's Inventing on Principle presentation [45] demonstrated several live programming environments and served as inspiration for later work [19, 24]. Babylonian-style Programming [36] explored how to manage multiple examples inline with the source code for live execution—individual examples could be switched on and off, an interaction we could adopt in *Maniposynth* to selectively reduce the number of values shown in the function IO grids.

**In-editor PBE/PBD** Like *Maniposynth*'s programming by examples (PBE) synthesizer, recent work has begun to explore offering PBE and PBD interactions within a traditional, textual programming environment.

Several systems generate code within a computational notebook via manipulations of visualized values. Wrex [5] adapts the FlashFill [12] PBE workflow to Pandas dataframes in Jupyter notebooks—after demonstrating examples of a desired data transformation in a dataframe spreadsheet view, Wrex outputs readable Python code. Similarly, the PBD systems B2 [47], mage [47], and Mito [4] transform step-by-step interactions with displayed notebook values into Python code in the notebook.

In the context of ordinary IDEs, CodeHint [10] and SnipPy [8] provide program synthesis interactions in the live context of the user's incomplete code. With CodeHint, users set a breakpoint in their Java program and describe some property about a value they want—CodeHint will enumerate method calls in the dynamic execution environment at the breakup to find a satisfying expression. Like *Maniposynth*, CodeHint leverages a statistics model to rank results. Notably, users with CodeHint were significantly faster and more successful at completing given tasks than users without. For Python, SnipPy [8] adapts the Projection Boxes tabular display of live program values [24] to perform PBE in the context of live Python values. Through their user study, the authors identified the *user-synthesizer gap*: the difference in the user's expectations of what the synthesizer can do versus what it actually can do. This gap phenomena did appear in our study as well. P2 spent considerable time writing many asserts to try to synthesize list `contains`, whose output is of type `bool`. This was before *Maniposynth* had a statistics model and synthesizers are excellent at finding superfluous ways to produce `true` and `false`, so P2 had little success. Even so, synthesis was new to our participants, so our study does not provide evidence on how persistent any understanding gap might be.

**Bidirectional, bimodal programming** Some systems represent programs as ordinary text, but also allow direct manipulation on program *outputs* to be back-propagated to change the original code. Usually, these changes are "small" changes to literals in the program—such as numbers [2, 21, 27, 9], strings [46, 39, 21, 27], or lists [27]. More full-featured program construction via output manipulations is available in a few systems for programs that output graphics [14, 29, 38].

Although *Maniposynth* also centers values as subjects for manipulation, we do not yet apply bidirectional techniques to deeply back-propagate a change on a value—direct changes on a value are only allowed when the value was introduced as a literal in the immediately associated expression. An earlier version of ***Maniposynth*** did have limited back-propagation abilities, but we disabled these when we noticed they caused trouble in the user study—manipulation on a value would inconspicuously change a literal in a very different part of the program. Determining an understandable meaning of such direct changes on a value remains an avenue for future work.

## 6 Future Work and Conclusion

How close is ***Maniposynth*** to achieving its goals of providing a value-centric, non-linear programming environment? Based on the examples we implemented and feedback from our study participants, ***Maniposynth*** largely succeeded at providing useful live values. The non-linear features functioned moderately well—users rarely had to think about binding order—but ***Maniposynth*** was not immediately learnable and would benefit from more explicit labeling of the various kinds of elements on the canvas. Through our observations, we hypothesize that expression-oriented and value-oriented modes of thinking are distinct states of mind, and experienced programmers tend towards the former. An intriguing possibility for future work is to experimentally validate that expression-oriented and value-oriented thinking are actually modes—*i.e.* the activity of considering values discourages considering expressions, and vice versa. More immediately, there are possible changes to ***Maniposynth*** that might encourage more value-focused interaction.

One experiment we would like to try is to change the display of variable uses so that, instead of the name of the variable, the current value of the variable is shown instead, with the name as a tooltip or subscript. This change might nudge users out of the expression-oriented mode of thinking back towards value-oriented thinking.

An intriguing corollary experiment was requested by P1. To keep track of where values came from, P1 wanted values to be drawn with unique colors all the time, rather than only when the autocomplete options were open. We would like to explore this as well.

Finally, while dragging items onto an expression is quite useful, in the current version of ***Maniposynth*** dragging items onto values is less-so. When working through the examples, the implementor dragged some item onto a value on only 4 occasions; dragging onto an expression happened 209 times. In the future, dragging a value to a value might open a menu of possible ways to combine the values, further increasing the utility of interacting with values.

In conclusion, Eros [6] showed that non-linearity complements functional programming, ***Maniposynth*** showed that non-linearity can be maintain even when the program is ordinary code, though the current ***Maniposynth*** is somewhat more expression-centric than Eros. The above are possible ways ***Maniposynth*** might become more value-centric. Our overall goal is to make programming feel like a tangible process of molding and forming. ***Maniposynth*** points a way forward to that goal.

## References

1 Andrew Begel. LogoBlocks: A Graphical Programming Language for Interacting with the World, 1996. Advanced Undergraduate Project, MIT Media Lab.

2 Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and Direct Manipulation, Together at Last. In *Conference on Programming Language Design and Implementation (PLDI)*, 2016.

**3** Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration.* MIT Press, 1993.

**4** Jacob Diamond-Reivich. Mito: Edit a Spreadsheet. Generate Production Ready Python. In *LIVE Workshop*, 2020. URL: `https://liveprog.org/live-2020/Mito-Edit-a-Spreadsheet-Generate-Production-Ready-Python/`.

**5** Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. Wrex: A Unifed Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. *Conference on Human Factors in Computing Systems (CHI)*, 2020.

**6** Conal Elliott. Tangible Functional Programming. In *International Conference on Functional Programming (ICFP)*, 2007. `http://conal.net/papers/Eros/`.

**7** Enso. Enso. URL: `https://enso.org/`.

**8** Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Small-Step Live Programming by Example. In *Symposium on User Interface Software and Technology (UIST)*, 2020.

**9** Koumei Fukahori, Daisuke Sakamoto, Jun Kato, and Takeo Igarashi. CapStudio: An Interactive Screencast for Visual Application Development. In *Conference on Human Factors in Computing Systems (CHI), Extended Abstracts*, 2014.

**10** Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *International Conference on Software Engineering (ICSE)*, 2014.

**11** Thomas R. G. Green and Marian Petre. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *J. Vis. Lang. Comput.*, 7(2):131–174, 1996. `doi:10.1006/jvlc.1996.0009`.

**12** Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Symposium on Principles of Programming Languages (POPL)*, 2011.

**13** Philip J. Guo. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Technical Symposium on Computer Science Education (SIGCSE)*, 2013.

**14** Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Symposium on User Interface Software and Technology (UIST)*, 2019.

**15** Joshua Horowitz. PANE: Programming with Visible Data. In *LIVE Workshop*, 2018. `http://joshuahhh.com/projects/pane/`.

**16** Yen-Teh Hsia and Allen L. Ambler. Programming Through Pictorial Transformations. In *Internation Conference on Computer Languages*, 1988.

**17** Christopher D. Hundhausen and Jonathan Lee Brown. What You See Is What You Code: A live Algorithm Development and Visualization Environment for Novice Learners. *Journal of Visual Languages and Computing*, 2007. `doi:10.1016/j.jvlc.2006.03.002`.

**18** Ruyi Ji, Yican Sun, Yingfei Xiong, and Zhenjiang Hu. Guiding Dynamic Programing Via Structural Probability for Accelerating Programming by Example. *Proc. ACM Program. Lang.*, 4(OOPSLA):224:1–224:29, 2020. `doi:10.1145/3428292`.

**19** Saketh Kasibatla and Alex Warth. Seymour: Live Programming for the Classroom. In *LIVE Workshop*, 2017.

**20** Andrew J. Ko and Brad A. Myers. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Human Factors in Computing Systems (CHI)*, 2006.

**21** Kevin Kwok and Guillermo Webster. Carbide Alpha, 2016. `https://alpha.trycarbide.com/`.

**22** A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

**23** Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating Search-based Program Synthesis using Learned Probabilistic Models. In *Conference on Programming Language Design and Implementation (PLDI)*, 2018.

**24** Sorin Lerner. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. *Conference on Human Factors in Computing Systems (CHI)*, 2020.

**25** H. Lieberman, editor. *Your Wish is My Command: Programming by Example.* Morgan Kaufmann Publishers Inc., 2001.

**26** Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.*, 4(ICFP):109:1–109:29, 2020. `doi:10.1145/3408991`.

**27** Mikaël Mayer, Viktor Kunčak, and Ravi Chugh. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2018.

**28** Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. User Interaction Models for Disambiguation In Programming by Example. In *Symposium on User Interface Software and Technology (UIST)*, 2015.

**29** Sean McDirmid. A Live Programming Experience. In *Future Programming Workshop, Strange Loop*, 2015. `https://onedrive.live.com/download?cid=51C4267D41507773&resid=51C4267D41507773%2111492&authkey=AMwcxdryTyPiuW8` `https://www.youtube.com/watch?v=YLrdhFEAiqo`.

**30** Francesmary Modugno, Albert T. Corbett, and Brad A. Myers. Graphical Representation of Programs In a Demonstrational Visual Shell - An Empirical Evaluation. *ACM Trans. Comput. Hum. Interact.*, 4(3):276–308, 1997. `doi:10.1145/264645.264659`.

**31** National Instruments. Labview. `https://www.ni.com/en-us/shop/labview.html`.

**32** Gabriel Scherer Naëla Courant, Julien Lepiller. camlboot, 2020. URL: `https://github.com/Ekdohibs/camlboot/`.

**33** Tobias Nipkow and Mohammad Abdulaziz. Functional Data Structures (in2347), 2020. Technische Universität München. URL: `https://github.com/nipkow/fds_ss20/tree/daae0f92277b0df86f34ec747c7b3f1c5f0a725c`.

**34** Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.*, 3(POPL):14:1–14:32, 2019. `doi:10.1145/3290327`.

**35** Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis. In *Conference on Programming Language Design and Implementation (PLDI)*, 2015.

**36** David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style Programming - Design and Implementation of An Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.*, 3(3):9, 2019. `doi:10.22152/programming-journal.org/2019/3/9`.

**37** Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Communications of the ACM (CACM)*, 2009.

**38** Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. *Transmorphic: Mapping Direct Manipulation to Source Code Transformations.* 2017.

**39** Christopher Schuster and Cormac Flanagan. Live Programming by Example: Using Direct Manipulation for Live Program Synthesis. In *LIVE Workshop*, 2016.

**40** Paul Shen. natto.dev. URL: `https://natto.dev/`.

**41** Ben Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, August 1983.

**42** David Canfield Smith. *Pygmalion: A Creative Programming Environment.* PhD thesis, Stanford University, 1975.

**43** Fabio Spampinato. Highlight VS Code Extension, 2021. URL: `https://marketplace.visualstudio.com/items?itemName=fabiospampinato.vscode-highlight`.

**44** William Robert Sutherland. *The On-line Graphical Specification of Computer Procedures.* PhD thesis, Massachusetts Institute of Technology, 1966.

**45** Victor, Bret. Inventing on Principle, 2012. `https://vimeo.com/36579366`.

**46** Xiaoyin Wang, Lu Zhang, Tao Xie, Yingfei Xiong, and Hong Mei. Automating Presentation Changes in Dynamic Web Applications via Collaborative Hybrid Analysis. In *International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

**47** Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. B2: Bridging Code and Interactive Visualization In Computational Notebooks. In *Symposium on User Interface Software and Technology (UIST)*, 2020.


# A How to Configure Vim and Emacs to Auto-reload Changed Files

*Maniposynth* writes updated code directly to the file on disk and expects the user's text editor will update automatically. Vim and Emacs do not auto-reload files from disk by default, but can be configured to do so.

## A.1 Vim

Thanks to eli on Super User[7] for this solution. Run this in Vim:

```
:set autoread | au CursorHold * checktime | call feedkeys("lh")
```

After cursor stops moving, this will check every `updatetime` seconds for file changes, which is every 4 seconds by default. (More specifically, Vim waits for the cursor to stop moving for some time, then checks disk, then moves the cursor again with "lh" to retrigger and loop.)

To poll every half second instead of every 4 seconds:

```
:set updatetime=500
```

If you get annoying bells, turn Vim's bell off:[8]

```
:set visualbell t_vb=
```

## A.2 Emacs

For Emacs, enable `global-auto-revert-mode`:

1. Hit F10 to go to the top menu.
2. Navigate Options > Customize Emacs > Specific Option.
3. Type `global-auto-revert-mode`
4. Navigate cursor to `[ Toggle ]` and hit Enter.
5. Navigate to `[ Apply and Save ]` and hit Enter.

[7] `https://superuser.com/a/1286322`
[8] `https://unix.stackexchange.com/a/5313`