It's Alive! Continuous Feedback in UI Programming

Sebastian Burckhardt Manuel Fähndrich
Peli de Halleux Sean McDirmid
Michal Moskal Nikolai Tillmann
Microsoft Research
{sburckha,maf,jhalleux,smcdirm,micmo,nikolait}@microsoft.com

Jun Kato
The University of Tokyo
i@junkato.jp

Abstract

Live programming allows programmers to edit the code of a running program and immediately see the effect of the code changes. This tightening of the traditional edit-compile-run cycle reduces the cognitive gap between program code and execution, improving the learning experience of beginning programmers while boosting the productivity of seasoned ones. Unfortunately, live programming is difficult to realize in practice as imperative languages lack well-defined abstraction boundaries that make live programming responsive or its feedback comprehensible.

This paper enables live programming for user interface programming by cleanly separating the rendering and non-rendering aspects of a UI program, allowing the display to be refreshed on a code change without restarting the program. A type and effect system formalizes this separation and provides an evaluation model that incorporates the code update step. By putting live programming on a more formal footing, we hope to enable critical and technical discussion of live programming systems.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]; D.2.2 [*Design Tools and Techniques*]: User Interfaces; D.2.6 [*Programming Environments*]

General Terms Languages, Human Factors

Keywords Live Programming, Graphical User Interface

1. Introduction

One major difficulty in writing programs is that a programmer must effectively simulate parts of the execution of the program in his mind during development [14, 19]. If we can narrow the gap between the program text and seeing how the program behaves, productivity during code editing and debugging could be improved substantially. This gap takes two forms: 1) the time gap between making an edit and seeing the effect of the change is dominated by rebuilding the program, executing the program and guiding it (possibly with manual input) to the place where the edit can be observed. Speeding up this "edit-compile-run" cycle can narrow the time gap, but the re-execution part remains critical and cannot be narrowed by improvements in development tools. 2) the perception gap is the cognitive distance between looking at the code and understanding what the code will do when it is run. Addressing this perception gap requires an environment where code can be edited continuously with uninterrupted "live" feedback, showing the consequences of those edits. This so-called *live programming* [11] enables more fluid problem solving compared to today's common "edit-compile-debug" style of programming.

Most people are familiar with live programming in the context of spreadsheets, where data and formulae can be edited and the effect of those edits be seen immediately. Besides spreadsheets, languages based on mostly declarative programming models, including many visual languages like Pure Data [20], already provide a live programming experience, but these languages are not expressive enough for more complex general purpose programs [6].

Grafting live programming features onto existing mainstream imperative programming languages such as Java or C# is daunting. Currently, such languages support fix-and-continue features that allow a restricted set of code changes to a live program and then continue execution. However, fix-and-continue alone does not provide a live programming experience. Live programming requires at a minimum some state snapshot, some re-execution of changed code, and some re-displaying of results. Exactly what data to save, what code to re-execute, and which parts of the UI to maintain is difficult to answer in a general purpose language. A natural starting point is re-execution of a trace of the entire program to the current point. However, apart from the cost of trace capturing and re-execution, traces are problematic since code changes can cause the re-execution to diverge from the previous trace. In light of this observation, it seems difficult to establish a precise technical definition of live programming to start with.

This paper tackles the question of live programming in the domain of user interface (UI) programming by proposing a formal model, where a program consists of both code and persistent data. The program is event based and viewed as continually executing. A code change is simply one possible transition of the program in our model. The benefit of formalizing such a language and execution model is that it unambiguously answers questions about what state is saved, what state is rebuilt, and what code is re-executed as a result of a code change. Our language is imperative and has an explicit imperative way of building up user-interface state in the form of nested *boxes*, akin to TeX and HTML, on a stack of pages. We separate both the UI state from ordinary state, and the render code that builds UI state from ordinary code. As a result, upon code changes, we throw away the UI state, and then rebuild the UI state for the currently displayed page using the separated render code for that page.

Beyond achieving live feedback, our model also reduces the perceptive gap between code and the UI components rendered on a page by the use of a `boxed` construct that builds a UI element as a side effect. Using syntactic nesting and arbitrary code such

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06. . . $15.00

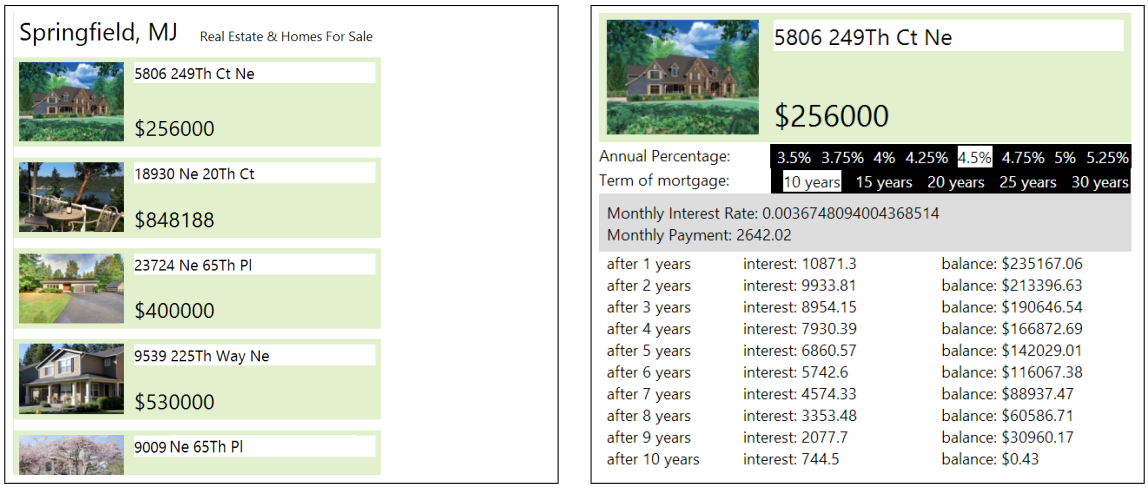

Figure 1. The two pages of the mortgage calculator example; a start page is on the left; a detail page is on the right.

as loops, conditionals, and procedure abstraction, UI construction is very expressive and not limited to a specialized declarative sub-language. We make the following contributions:

- We propose a programming model where UI state is built using procedural code, but maintained separate from the normal state of the program. UI's are organized as page stacks with operations to push a new page or pop the current page.
- We put live programming on a formal footing by providing an execution model in which one can argue what parts of the code are re-executed upon a program change.
- Although live programming has recently been demonstrated [10, 27] to much fanfare, designing and building these systems is so far a black art. We report on our experience implementing and using our programming model in TouchDevelop [1, 25], a device independent browser-based programming language and development environment.

Section 2 further motivates live programming while Section 3 introduces how our language and environment supports it. Section 4 describes and formalizes our implementation, providing insights on how to design and build future systems. Section 5 presents our experience with our implementation and a discussion of the trade-offs made in its construction. Section 6 presents related work while Section 7 concludes.

2. Motivation

We now explain the motivation for live programming with a discussion of how it improves on existing programming practice. For this discussion, we use as a running example a simple program to browse local listings of houses for sale, and calculate mortgage payments and an amortization schedule. On startup, the application issues a web request to obtain listings that are displayed on the program's start page (Figure 1, left). When the user taps an entry, the program navigates to a detail page that displays the monthly mortgage payment and an amortization schedule (Figure 1, right). The user can modify the term of the mortgage and the annual percentage rate by tapping the corresponding box.

Suppose that the programmer is not satisfied with the UI and wishes to make the following fixes and improvements: (I1) adjust various margins to improve the visual appearance; (I2) print the monthly balance in properly formatted dollars and cents; and (I3) highlight every fifth line of the amortization schedule with a different color. If the code were written in a conventional language and programming environment, a programmer implementing the three improvements (I1-I3) would have to iterate the following steps:

1. Stop the program execution.
2. Discover and navigate to the code location that needs to be fixed.
3. Edit the code to change the behavior.
4. Compile the code, and restart the program.
5. Navigate to the appropriate UI context by waiting for the list to download, and clicking on an entry.
6. Inspect the display to verify the fix.
7. If not satisfied, continue with step 1.

This cycle is time-consuming and tiring since temporal delays drain programmer concentration; e.g., consider the waiting for code compilation or the list to download, navigating to the correct UI screen, and, while debugging, remembering procedure names and the call graph to retrieve relevant code. Hancock [11] compares this way of programming to the activity of an archer: "you aim, shoot, string another arrow, carefully correct the aim, shoot again, and so on." This productivity drain is particularly harmful when dealing with design properties that require many iterations in practice, such as margins, font sizes, or colors.

Conventional programming can currently be made more fluid in a couple of ways. First, a programmer could use a read-eval-print loop (REPL) to quickly experiment with code statements, such as monthly interest rate printing, on a command line. However, using a REPL precludes debugging the UI directly, and each feature must be developed as a command-line computation. Second, many IDEs for languages such as LISP [23], Smalltalk [9], Java, and C# support a "fix-and-continue" feature where the programmer can modify their code without restarting the debugging process. Unfortunately, fix-and-continue often does not result in responsive feedback: for the common "retained" UI where a program builds and modifies a tree of widget objects to be rendered, changing the code that initially builds this widget tree is meaningless as that code has already executed and will not execute again!

If editing and debugging a program were to occur simultaneously, the programmer could make an adjustment to interest rate printing, observe immediately what this change produces, and then

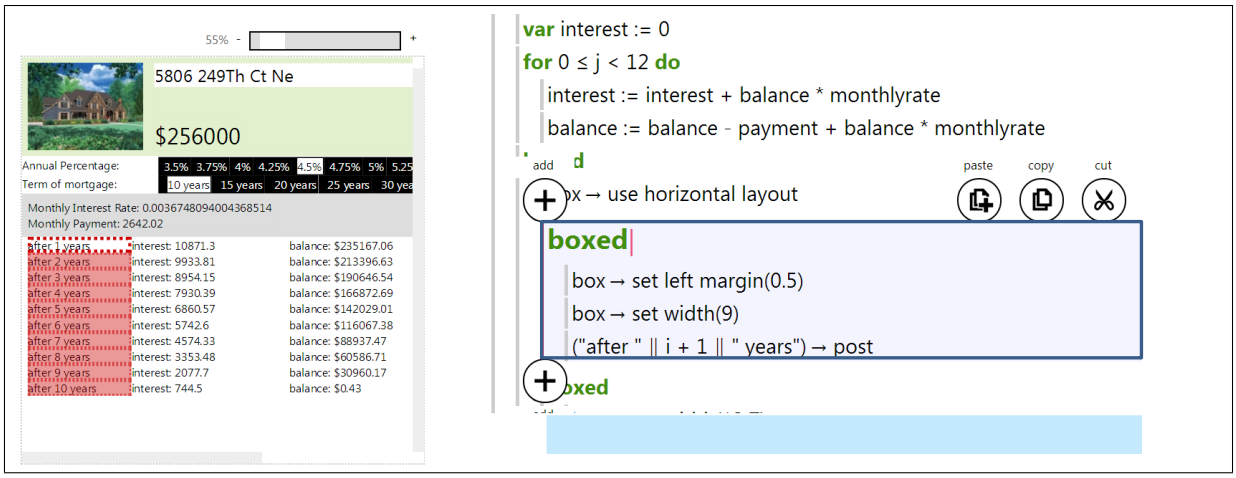

Figure 2. Code edited in the right *code view* is continuously type-checked, compiled, and executed, where its behavior is instantly visible in the left *live view*. Selecting a box in the left live view (highlighted red outline) causes the corresponding *boxed* statement to be selected in the right code view, and vice versa.

quickly make another edit to adjust the formatting based on that observation. Although parts of the program must be re-executed, the application and UI state of the program must be maintained so that the programmer can focus on editing and observing. Hancock [11] compares *live programming* to using a water hose in contrast to archery: "as we adjust which way the hose is pointed, the stream of water from us toward the target reflects back as an uninterrupted stream of information about the consequences of our aim."

Live programming has long been supported by many visual programming languages; e.g., a program in PureData [20] is expressed as a data flow graph that is continuously executing while the user is adding nodes and edges in this graph. Such visual languages realize live programming with inexpressive declarative programming models of simplified control flow constructs and heavily encapsulated state. For example, editing a PureData program simply causes re-execution of its data flow graph where any state encapsulated in a node is easily preserved.

Unfortunately, a large class of programs, including our running example, cannot easily be rewritten using existing declarative live visual languages given their inexpressive programming models. On the other hand, building an IDE that can support live programming for a more expressive language like C# is technically daunting given their support for complex control flow and imperative state. Many programming environments for conventional languages do offer a limited live experience by separating declarative style declarations from the imperative code that generates content; e.g., HTML and cascading style sheets (CSS) allow designers to change box structure and stylistic properties of web pages and inspect the result immediately while ignoring embedded Javascript code. This live experience can handle aesthetic tweaks like the I1 improvement in the list above, but unfortunately cannot support a wide variety of program alterations, such as the I2 and I3 improvements.

As observed in [17], updating a UI based on changes in the code is really not that different from updating the UI based on model changes. The important question is how to ensure that the view is up-to-date with respect to the model in the first place. The widely used model-view-controller (MVC) pattern (pioneered in [21]) requires the programmer to write code that reacts to model changes and performs the corresponding updates to the view. If the view is a complex function of the state, writing such code can be challenging (in database systems, this is known as the view-update problem), and it is questionable whether the programmer should be bothered to perform such dangerous mental acrobatics. We adopt a more straightforward solution: the programmer writes code that directly specifies how the view is constructed from the model, and just rerun this code whenever the model changes. We thus construct a fresh view instead of updating the existing one, known as the *immediate* vs. *retained* approach to GUI programming [18]. The immediate approach naturally enables live programming, because in response to a code change we can simply re-render the UI with the new code applied to the program's old state. The next section describes how we leverage this technique to enable live programming in our language.

3. Live Programming in TouchDevelop

We modify the TouchDevelop language [25] with programming model changes that enable live UI programming in an enhanced programming environment. The traditional edit-compile-run cycle is tightened and enhanced via the following three features:

Live Editing. The program keeps running while the programmer edits their code. Our editor provides a split screen that shows program execution on the left in the *live view*, and program code being edited on the right in the *code view* (Figure 2). All changes to UI-rendering code are immediately visible in the live view because the program is continuously being type-checked, compiled, and executed as the programmer edits.

UI-Code Navigation. The environment maintains navigable bi-directional connections between rendered elements in the live view and code that created these elements in the code view. If the user taps a UI element, which we call a *box*, in the live view, the editor on the right selects the **boxed** statement in the code view that created the UI element (Figure 2). Likewise, if the user selects a **boxed** statement in the code view, the corresponding box (or boxes) is selected in the live view. Note that a selected **boxed** statement appearing inside a loop corresponds to multiple boxes in the display, which are collectively selected in the live view (Figure 2).

Direct Manipulation. Combining both live editing and UI-code navigation, the programmer can directly change the attributes of a box in the live view, where the code view is updated automatically to reflect these changes (known as direct manipulation in [24]). For example, to insert a command to change

the size of a margin, the programmer can first select the corresponding box in the live view and then choose the margin property from a button menu, which inserts (if not present) a command in the code and positions the code cursor on the margin number. The programmer can then edit this number while observing the result in the live view.

Although one may imagine an implementation that provides some or all of these features without any changes to the programming language, such as by using deterministic replay or program analysis, we focus here on the programming language design rather than heroic tool construction. Thus, our main contribution is to demonstrate how to enable a simple and transparent implementation of a live experience by *specializing the programming model*.

We take concepts from UI programming—event-based programming, model-view separation, box-based layout, and page-stack navigation—and bake them directly into the execution model, language, and type system. We first present an example-based description of this model; an in depth description of the programming model appears in Section 4.

As usual, programmers can define global procedures and variables in TouchDevelop. In addition, programmers can now define *pages* that take arguments like procedures (an idea we borrowed from the Mobl programming language [13]). The TouchDevelop code for the start page of our example program in Section 2 is shown in Figure 3. Unlike procedures, pages have two bodies rather than one:

- The *initialization body* is executed before the page is rendered for the first time. It can update global variables, but it cannot create any UI elements (boxes). The initialization body for the **startup** page is the **init** code at the top in Figure 3. It downloads the list of properties for sale and stores it in the variable **listings**.
- The *render body* is called to build or refresh the display with interactive UI elements known as *boxes*; the **render** body in Figure 3 defines two top-level vertically-stacked boxes[1]. The first top-level box is the header. It contains two boxes of text laid out horizontally. The second top-level box contains all listings, where a box for each entry is created by iterating over each listing stored in the **listings** variable. Note that, while **render** code can read global variables, such as **listings**, it cannot write global variables. The only effects of render code are the construction of UI elements through **boxed** constructs.

In model-view terminology, the *view* of the program is thus defined by the **render** body of a page, while the *model* is expressed as the program's global variables. The **render** body is re-executed in response to changes to the model or to code called during rendering. The latter guarantees that the live view is always consistent with the current code. Since **render** code cannot modify global variables, re-execution of a **render** body preserves the model (values of global variables). Initialization of the model must occur in the initialization body, which is not automatically re-executed and therefore cannot undergo live programming.

Boxes, which are TouchDevelop's UI elements, are defined in a page as nested **boxed** statements in much the same way that DOM elements are defined in HTML. We refer to them collectively as a page's *box tree*. Unlike widgets in conventional UI libraries, boxes are not first-class values in TouchDevelop, meaning code does not manipulate the box tree structure directly. Instead, the box tree is created as a "side effect" of the execution weaving in and out of **boxed** statements. This execution is free to use loops, arbitrary conditionals, and procedure calls. For example, the last **boxed** statement in the code of Figure 3 is embedded in a for-loop,

[1] Vertical stacking is the default.

```
page startpage ()
init
    ◳ listings := ▷ getlistings()
    ◳ term := 20
    ◳ apr := 0.045
render
    boxed
        boxed
            box → use horizontal layout
            boxed
                box → set left margin(0.5)
                box → set font size(2.3)
                "Springfield, MJ" → post
            boxed
                box → set font size(1.3)
                box → set top margin(1.08)
                "Real Estate & Homes For Sale" → post
        boxed
            for 0 ≤ i < ◳ listings → count do
                boxed
                    ▷ display listentry(◳ listings → at(i))
                    on box → tapped do
                        push ▷ detailpage(◳ listings → at(i))
```

Figure 3. The source code for the start page of our mortgage calculator example, whose rendering was shown on the left of Figure 1 in Section 2. **Note on syntax:** attributes and methods are accessed via → rather than the typical `.`; global functions calls are preceded by ▷; and global variable references are preceded by a square symbol.

producing a number of boxes equal to the number of items in the **listings** variable. Additionally, it creates nested boxes by calling the **display listentry** procedure. Boxes have attributes that include layout parameters (margins, size, layout direction, font sizes), and event handlers.

Event handlers can be registered on boxes using the **on** statement to respond to interactions such as tapping or editing by the user. To respond to an event, event handlers can modify global variables as model state, or perform page nagivation such as popping the current page or pushing a new page. Handlers are not part of **render** code and are never executed as part of rendering. The last two lines in Figure 3 define a **tapped** event handler for each listing entry. When activated, it pushes the "detail" page for that entry, whose code is defined in Figure 4.

To maintain a clean separation between the model (global variables) and the view (boxes and **render** code) and to guarantee that the view is a well-defined function of the model, we enforce the following rules:

- The view is stateless: the display content (the box tree, including attributes and handlers) cannot be read by the code, and is discarded as soon as it becomes stale. Boxes are second class values and so cannot be referenced (aka aliased) outside rendering code. Attributes of a box can only be modified by statements inside the dynamic scope of the corresponding **boxed** statement.
- Render code can only read, but not modify global variables (model state).
- Non-render code, such as initialization bodies and event handlers, can modify global variables, but cannot produce boxes.

We show in Section 4 how we enforce these rules at compile time using a type and effect system.

```
page detailpage (
  listing : ⊞listing)
init
  do nothing
render
  boxed
    ▷ display listentry(listing)
  ▷ display apr menu
  ▷ display term menu
  var principal := listing → price → get
  var monthlyrate := math → exp(math → loge(1 + ◳apr) / 12) - 1
  var rr := math → pow(1 + monthlyrate, ◳term * 12)
  var payment := math → round with precision(principal * ((monthlyrate
  * rr) / (rr - 1)), 2)
  ▷ display results(monthlyrate, payment)
  boxed
    box → set font size(1)
    ▷ display amortization(principal, monthlyrate, payment)
```

Figure 4. The source code of the detail page.

This separation between model and view is sufficient for achieving the live editing feature of TouchDevelop's live programming experience. Code-execution reification and direct manipulation in turn are supported by TouchDevelop's simple box model, which allows the environment to easily keep track of code/box mappings. Our enhanced programming model also helps to keep UI code concise and readable. Although the code in Figure 3 *looks* a bit like declarative code, it executes just like any other procedural code: it is free to use loops to create multiple boxes, procedural abstraction to keep the code organized, and conditional statements to customize the display based on arbitrary conditions.

3.1 Example Improvements

We now discuss how the three improvements (I1–I3) described at the beginning of Section 2 are applied by the programmer in the live programming enhanced version of TouchDevelop. The programmer starts the program and begins editing our example program's **startpage**, whose code is shown in Figure 3. The first improvement made is to tweak the margins of the page (I1), which is done through direct manipulation: one simply selects the box to modify in the output display which brings up a menu on the code side to change it.

When the programmer taps an entry on the start page, its detail page opens (Figure 2) and the code view shows the code in Figure 4. The monthly amortization table of this page is generated by calling the **display amortization** function defined in Figure 5. Selecting one of the "balance" cells in the detail page selects the corresponding **boxed** statement in the code view (last **boxed** in Figure 5. Changing this statement to:

```
boxed
  var dollars := math → floor(balance)
  var cents := math → round((balance - dollars) * 100) || ""
  if cents → count < 2 then
    cents := "0" || cents
  ("balance: $" || dollars || "." || cents) → post
```

will cause the balance to print correctly in dollars and cents, implementing our I2 improvement; balance printing is updated for all amortization table rows as soon as we complete the last line of this modification.

The last improvement, high-lighting every fifth year of the amortization in another color (I3), involves selecting an amortization row in in the live view, which causes the first top-level box

```
private action display amortization (
  balance : Number,
  monthlyrate : Number,
  payment : Number)
do
  for 0 ≤ i < ◳term do
    var interest := 0
    for 0 ≤ j < 12 do
      interest := interest + balance * monthlyrate
      balance := balance - payment + balance * monthlyrate
    boxed
      box → use horizontal layout
      boxed
        box → set left margin(0.5)
        box → set width(9)
        ("after " || i + 1 || " years") → post
      boxed
        box → set width(13.7)
        ("interest: " || math → round with precision(interest, 2)) → post
      boxed
        box → set width(13)
        ("balance: $" || math → round with precision(balance, 2))
        → post
```

Figure 5. The procedure to display the amortization schedule.

of Figure 5 to be selected in the code view. The programmer can add the code

```
if math → mod(i, 5) == 4 then
  box → set background(colors → light blue)
```

to set the background of every fifth row to a light blue color. This improvement demonstrates how our live programming improves on UI editors that operate over purely declarative languages. E.g., HTML/CSS has special support for even-odd coloring, but cannot go beyond that without the loss of live feedback.

4. Formal Model

In this section, we develop a formal operational model. This model removes any ambiguities that are often a source of confusion in reactive systems, where user actions (back button, box tapping, code updates) are interleaved with program execution (event handling, rendering). It also clarifies how we avoid some typical problems, such as (1) the programmer unintentionally violating the view-model separation, or (2) strange failures due to inconsistencies between various versions of the code. We start with the syntax for writing code (Section 4.1). Then, we present an operational model that defines how the system state (which includes the code) evolves when handling user events, receiving code updates, or performing execution steps (Section 4.2). Finally, we present and discuss our type and effect system (Section 4.3). We do not formalize the visual layout of box trees.

4.1 Expression Syntax

We show the expression syntax in Fig. 6. It is based on the simply typed lambda calculus with the following additions:

– We use tuples to simplify the passing of multiple values to functions and page code. Also, empty tuples serve as the unit value, which we use heavily (since many of our operations are imperative in nature).

– We use global variables (representing the model state) and global function definitions (representing the current code). This

Identifiers:

g	::=	...	global variables
f	::=	...	global functions
p	::=	start | ...	page names
a	::=	ontap | margin | ...	box attributes
μ	::=	p | r | s	pure, render, state effect

Types:

τ	::=	number	(number)
	|	string	(string)
	|	$(\tau_1, ..., \tau_n)$	(tuple), $(n \geq 0)$
	|	$\tau \xrightarrow{\mu} \tau$	(function)

Environment:

Γ	::=	ϵ	
	|	$\Gamma, x : \tau$	(variable type)
	|	$\Gamma, a : \tau$	(attribute type)

Values:

v	::=	n	(number literal)
	|	s	(string literal)
	|	x	(variable)
	|	$(v_1, ..., v_n)$	(tuple)
	|	$\lambda(x : \tau).\, e$	(lambda)

Expressions:

e	::=	v	(value)
	|	$e_1 e_2$	(application)
	|	f	(function)
	|	$(e_1, ..., e_n)$	(tuple), $(n \geq 0)$
	|	$e.n$	(projection), $(n \geq 1)$
	|	g	(read global)
	|	$g := e$	(write global)
	|	push p e	(push new page)
	|	pop	(pop page)
	|	boxed e	(create box)
	|	post e	(post content)
	|	box.$a := e$	(set box attribute)

Evaluation Contexts:

E	::=	$[] \mid E\ e \mid v\ E$
	|	$(v_1, ..., v_i, E, e_j, ..., e_n) \mid E.n \mid g := E$
	|	push p E | post E | box.$a := E$

Figure 6. The syntax of types, values, expressions, and evaluation contexts.

separation is important during code updates, when we need to fix up the model state and purge stale code.

- We use page names to identify pages, and we support operations for pushing a new page (passing a parameter) and popping the current page.
- We support boxed statements to create a box, post e to add content to the current box, and box.$a := e$ to modify an attribute of the current box.

Our calculus is intentionally kept concise even though our examples use a higher level syntax. Loops are expressible in our calculus via recursion through global functions, conditionals via lambda abstractions and thunks. Handlers on boxes are simply attributes that can be set to lambda expressions. These differences are purely syntactic and no expressivity is lost.

System State:

σ	::=	(C, D, S, P, Q)

System Components:

C	::=	$\epsilon \mid C\ d$	(program code)
D	::=	$\perp \mid B$	(display)
S	::=	$\epsilon \mid S[g \mapsto v]$	(store)
P	::=	$\epsilon \mid P\ (p, v)$	(page stack)
Q	::=	$\epsilon \mid Q\ q$	(event queue)

Program Definitions:

d	::=	global $g : \tau = v$	(global)
	|	fun $f : \tau$ is e	(function)
	|	page $p(\tau)$ init e_1 render e_2	(page)

Box Content:

B	::=	ϵ	(empty)
	|	$B\ v$	(leaf content)
	|	$B\ [a = v]$	(box attribute)
	|	$B\ \langle B \rangle$	(nested box)

Events:

q	::=	[exec v]	(execute thunk)
	|	[push p v]	(push new page)
	|	[pop]	(pop page)

Figure 7. Definitions for system states, programs, box content, and events.

4.2 System Model

Our operational model is defined by a set of system states and a set of system transitions. A system state is a tuple (C, D, S, P, Q) as defined in Fig. 7. Note that for simplicity, we represent our data structures as sequences (with ϵ being the empty sequence), though an actual implementation would use specialized data structures such as maps, sets, queues, and so on. The meaning of the components is as follows:

- C represents the code (i.e. the program). It contains (1) global variable definitions that specify a name, a type, and an initial value, (2) function definitions that specify a name, a function type, and a lambda expression, and (3) page definitions that specify a page name, the type of the argument passed to the page on construction, and two functions to respectively be called on initialization and rendering. To save space, we write $C(p) = (f_i, f_r)$ as a shorter form for (page $p(\tau)$ init f_i render f_r) $\in C$.
- D represents what is currently displayed to the user. It contains either box content B (recursively defined as a sequence of layout attributes, values, and nested boxes), or the special value $\perp$ to indicate that the display is stale and needs to be refreshed.
- S represents the store (i.e. the values of global variables). We represent S as a sequence of key-value pairs $[g \mapsto v]$; the rightmost occurrence of a key g defines its current value, denoted $S(g)$.
- P represents the page stack. It is a sequence of pairs (p, v) where p is a page identifier and v is the argument value that was supplied when the page was created. We add and remove entries at the end of the sequence.
- Q represents the event queue. It contains three kinds of events: [exec v], [push p x], [pop]. We enqueue by adding elements

Pure execution steps:

$$\text{(EP-FUN)}\ \frac{(\mathsf{fun}\ f : \tau\ \mathsf{is}\ e) \in C}{(C, S, E[f]) \to_{\mathsf{p}} (C, S, E[e])}$$

$$\text{(EP-APP)}\ \frac{}{(C, S, E[\lambda(x : \tau).e)\ v]) \to_{\mathsf{p}} (C, S, E[e[v/x]])}$$

$$\text{(EP-TUPLE)}\ \frac{}{((C, S, E[(v_1, ..., v_m).n]) \to_{\mathsf{p}} (C, S, E[v_n])}$$

$$\text{(EP-GLOBAL-1)}\ \frac{S(g) = v}{((C, S, E[g]) \to_{\mathsf{p}} (C, S, E[v])}$$

$$\text{(EP-GLOBAL-2)}\ \frac{g \notin dom\ S \qquad \mathsf{global}\ g : \tau = v \in C}{((C, S, E[g]) \to_{\mathsf{p}} (C, S, E[v])}$$

Execution steps for standard mode:

$$\text{(ES-PURE)}\ \frac{(C, S, e) \to_{\mathsf{p}} (C, S, e')}{(C, S, Q, e) \to_{\mathsf{s}} (C, S, Q, e')}$$

$$\text{(ES-ASSIGN)}\ \frac{}{(C, S, Q, E[g := v]) \to_{\mathsf{s}} (C, S[g \mapsto v], Q, E[()])}$$

$$\text{(ES-PUSH)}\ \frac{}{(C, S, Q, E[\mathsf{push}\ p\ v]) \to_{\mathsf{s}} (C, S, [\mathsf{push}\ p\ v]\ Q, E[()])}$$

$$\text{(ES-POP)}\ \frac{}{(C, S, Q, E[\mathsf{pop}]) \to_{\mathsf{s}} (C, S, [\mathsf{pop}]\ Q, E[()])}$$

Execution steps for render mode:

$$\text{(ER-PURE)}\ \frac{(C, S, e) \to_{\mathsf{p}} (C, S, e')}{(C, S, B, e) \to_{\mathsf{r}} (C, S, B, e')}$$

$$\text{(ER-POST)}\ \frac{}{((C, S, B, E[\mathsf{post}\ v]) \to_{\mathsf{r}} (C, S, B\ v, E[()])}$$

$$\text{(ER-ATTR)}\ \frac{}{((C, S, B, E[\mathsf{box}.a := v]) \to_{\mathsf{r}} (C, S, B\ [a = v], E[()])}$$

$$\text{(ER-BOXED)}\ \frac{(C, S, \epsilon, e) \to_{\mathsf{r}}^{*} (C, S, B', v)}{(C, S, B, E[\mathsf{boxed}\ e]) \to_{\mathsf{r}} (C, S, B\ \langle B' \rangle, E[v])}$$

Figure 8. Expression evaluation steps are defined by $\to_{\mathsf{p}}$, $\to_{\mathsf{s}}$ and $\to_{\mathsf{r}}$ for pure mode, standard mode, and render mode, respectively. $\to_{\mu}^{*}$ is the reflexive transitive closure of $\to_{\mu}$.

to the left of the sequence, and dequeue by removing elements from the right end of the sequence.

At the heart of our operational semantics are the small expression evaluation steps, defined in Fig. 8. We distinguish three different steps based on their effects. Pure steps are side-effect free, but may depend on the code C and the current global state S, thus they are of the form $(C, S, e) \to_{\mathsf{p}} (C, S, e')$. Standard execution steps are of the form $(C, S, Q, e) \to_{\mathsf{s}} (C, S', Q', e')$; they may modify the state S or add elements to the event queue Q. Render steps are of the form $(C, S, B, e) \to_{\mathsf{r}} (C, S, B', e')$; they may be pure, or can append an element to the current box content B.

Our evaluation rules enforce that render functions have no side effects other than updating the display, and that the display cannot be accessed by any other user code. This is important, because it is otherwise much too easy for programmers to (intentionally or unintentionally) break the principles of our model-view separation, with highly confusing consequences.

At the system level, We define the *system step relation* $\to_{\mathsf{g}}$ as a binary relation on system states in Fig. 9. We call a system state *stable* if the event queue is empty, and the page stack is non-empty. In stable states, the system is waiting for user actions such as (TAP) for tapping a box in the display, (BACK) for hitting the back button, and (UPDATE) for code updates.

Three rules that enqueue events:

$$\text{(STARTUP)}\ \frac{}{(C, D, S, \epsilon, \epsilon) \to_{\mathsf{g}} (C, \bot, S, \epsilon, [\mathsf{push\ start}\ ()])}$$

$$\text{(TAP)}\ \frac{[\mathsf{ontap} = v] \in B}{(C, B, S, P, Q) \to_{\mathsf{g}} (C, \bot, S, P, [\mathsf{exec}\ v]\ Q)}$$

$$\text{(BACK)}\ \frac{}{(C, D, S, P, Q) \to_{\mathsf{g}} (C, \bot, S, P, [\mathsf{pop}]\ Q)}$$

Three rules that handle events:

$$\text{(THUNK)}\ \frac{(C, S, Q, v\ ()) \to_{\mathsf{s}}^{*} (C, S', Q', ())}{(C, D, S, P, Q\ [\mathsf{exec}\ v]) \to_{\mathsf{g}} (C, \bot, S', P, Q')}$$

$$\text{(PUSH)}\ \frac{C(p) = (f_i, f_r) \qquad (C, S, Q, (f_i\ v)) \to_{\mathsf{s}}^{*} (C, S', Q', ())}{(C, D, S, P, Q\ [\mathsf{push}\ p\ v]) \to_{\mathsf{g}} (C, \bot, S', P\ (p, v), Q')}$$

$$\text{(POP)}\ \frac{P = P'(p, v) \quad \text{or} \quad P = P' = \epsilon}{(C, D, S, P, Q\ [\mathsf{pop}]) \to_{\mathsf{g}} (C, \bot, S, P', Q)}$$

One rule to refresh the display:

$$\text{(RENDER)}\ \frac{C(p) = (f_i, f_r) \qquad (C, S, \epsilon, (f_r\ v)) \to_{\mathsf{r}}^{*} (C, S, B, ())}{(C, \bot, S, P\ (p, v), \epsilon) \to_{\mathsf{g}} (C, B, S, P\ (p, v), \epsilon)}$$

One rule to change the program code:

$$\text{(UPDATE)}\ \frac{C' \vdash C' \qquad C' : S \triangleright S' \qquad C' : P \triangleright P'}{(C, D, S, P, \epsilon) \to_{\mathsf{g}} (C', \bot, S', P', \epsilon)}$$

Figure 9. System steps are defined by $\to_{\mathsf{g}}$.

We define the initial system state to be $(C, \bot, \epsilon, \epsilon, \epsilon)$, which is unstable. While the system state is unstable, one of the following transitions is always enabled:

- If the page stack is empty, we can perform the transition (STARTUP) which enqueues an event $[\mathsf{push\ start}\ ()]$ causing the system to create the start page.
- If the event queue is not empty, we can dequeue the next element with one of the following transitions:
  - (THUNK) dequeues $[\mathsf{exec}\ v]$ and executes the thunk v (which is a lambda function that takes a unit value and returns a unit value). It executes v in standard execution mode, taking as many small steps as necessary to reduce the expression $v\ ()$ to a value.
  - (PUSH) dequeues $[\mathsf{push}\ p\ v]$, pushes a new page (p, v) onto the page stack, and executes the page initialization code (passing v as the argument), in standard execution mode. It takes as many small steps as necessary to reduce to a value.
  - (POP) dequeues $[\mathsf{pop}]$, and either pops the top page, or does nothing (if the page stack is already empty).

Some of these transitions can enqueue more events onto the queue (for example, executing a push or pop expression in user code enqueues a push or pop event). This can lead to an infinite loop of pushing new pages. Also, the execution of user code may of course diverge. Apart from those nonterminating cases, however, we eventually reach a stable state (we never get stuck, as discussed in Section 4.3). Thus the system is always live, either in an active state (executing some user code), or in a stable state (ready to handle user events or code updates).

All global transitions, except for (RENDER), also invalidate the display (set it to $\perp$). The display remains invalid until we do the (RENDER) transition. This transition executes the render code for the page that is currently at the top of the stack, in render mode, taking as many small steps as necessary. This mechanism guarantees that the display is never stale, but either invalid or current with respect to the model state and the code. In particular, it is not possible to activate tap handlers on a stale display: the prerequisite of the rule (TAP) can only be satisfied if the display is valid, which also implies that Q is empty.

In a stable state, the transition (UPDATE) allows the user to update the code (swap new code C' for old code C). Its first prerequisite (to be formally defined in Section 4.3) is $C' \vdash C'$, which means that C' must be well-typed and satisfy a number of sanity conditions. Note that there is no requirement that C' is related in any way to C. Instead, our transition performs a fix-up of the global state (defined by the relations $C' : S \triangleright S'$ and $C' : P \triangleright P'$ which we discuss in Section 4.3). Supporting arbitrary code changes is important in practice: limiting the changes a user can make is both complex to implement and explain, and unpleasant for the user.

Another important guarantee we make is that after a code update, the system contains no stale code (such as closures taken in earlier versions). The reason is that after applying rule (UPDATE), the display and the event queue are empty. Since neither global variables nor the page stack contain function values (we enforce this using the type system), the state contains no code.

4.3 Typing of Expressions and States

Fig. 10 shows how we type expressions. Our judgments have the form $C; \Gamma \vdash_\mu e : \tau$, meaning that we can type e as τ given the code C, context Γ, and effect μ (which is one of $\mathsf{p}, \mathsf{s}, \mathsf{r}$ for pure, state, or render). We define an attribute environment Γ_a that contains types for box attributes, such as $\mathsf{ontap} : () \xrightarrow{\mathsf{s}} ()$ and $\mathsf{margin} : \mathsf{number}$. Note that our model has an implicit top-level box, so render code can set attributes even outside a **boxed** statement.

The rules in Fig. 10 are mostly standard [15]. Rules indexed by effect variable μ can be instantiated to all three effect modes $\mathsf{p}, \mathsf{s}, \mathsf{r}$. The types relate to the operational semantics in that an expression e typable under effect μ reduces to a value under $\rightarrow_\mu^*$, thereby guaranteeing that render code can be reduced by $\rightarrow_\mathsf{r}$ rules, and that stateful code can be reduced by $\rightarrow_\mathsf{s}$ rules.

Fig. 11 shows how we type system states. Not surprisingly, it involves separate typing judgements for almost all system components. The top rule (T-SYS) ensures that there is a definition for the start page (otherwise we would be stuck before execution even starts). The typing judgments $C \vdash D$, $C \vdash S$, $C \vdash P$, and $C \vdash Q$ are straightforward typings for the display, global variables, page stack, and event queue. The typing judgments $C \vdash C$ enforce that no name is defined twice, and that definitions use correct typings. In particular, global variables must have function-free types (notated as $\rightarrow$-free), functions must be typable with the type they declare, and the render and init functions must be typeable under the corresponding effect.

Preservation. All small evaluation steps preserve the type of the evaluated expression (i.e. if $e \rightarrow_\mu e'$, we can type e' with the same type and effect as e), and leave the store and the queue well typed. This is a simple consequence of our type and effect system (we are using a widely known standard construction). System steps also preserve the typeability of the system state. This is mostly a simple consequence of manipulating the state correctly, and of the preservation guarantee for small evaluation steps. However, the (UPDATE) rule is interesting since it completely replaces the code. In this case, to ensure typeability of the state, we need to fix up the global state and page stack. The algorithm for this fix-up is shown in Fig. 12. Essentially, it just deletes whatever does not type.

$$\text{(S-EMPTY)}\ \frac{}{C : \epsilon \triangleright \epsilon} \qquad \text{(S-SKIP)}\ \frac{C : S \triangleright S' \quad g \notin C \vee (C; \epsilon \not\vdash_\mathsf{s} v : \tau)}{C : S\,[g \mapsto v] \triangleright S'}$$

$$\text{(S-OKAY)}\ \frac{C : S \triangleright S' \quad \mathsf{global}\ g : \tau =\in C \quad C; \epsilon \vdash_\mathsf{s} v : \tau}{C : S\,[g \mapsto v] \triangleright S'\,[g \mapsto v]}$$

$$\text{(P-EMPTY)}\ \frac{}{C : \epsilon \triangleright \epsilon} \qquad \text{(P-SKIP)}\ \frac{C : P \triangleright P' \quad p \notin C \vee (C; \epsilon \not\vdash_\mathsf{s} v : \tau)}{C : P\,(p, v) \triangleright P'}$$

$$\text{(P-OKAY)}\ \frac{C : P \triangleright P' \qquad C; \epsilon \vdash_\mathsf{s} v : \tau \quad \mathsf{page}\ p(\tau)\ \mathsf{init}\ e_1\ \mathsf{render}\ e_2 \in C}{C : P\,(p, v) \triangleright P'\,(p, v)}$$

Figure 12. Rules for fixing up the globals and the page stack.

Progress. Any expression e that is not a value and that types as $C; \Gamma \vdash_\mu e : \tau$, with $\mu = p$ or $\mu = s$, can take a step $e \rightarrow_\mu e'$. However, if $\mu = r$, progress may be only indirect: if $e = E[\mathsf{boxed}\ e']$ and e' has a diverging computation, then it is only e' that makes progress, but not e. At the system level, progress is also guaranteed with some restrictions. In a stable system state, the system makes no progress unless there are user-initiated actions. In unstable states, the system can always make progress except if there is a diverging expression evaluation (again, progress in that case is indirect: the expression evaluation makes progress, but the system as a whole does not).

5. Experience

We now present our experiences in designing, building, and using an enhanced version of TouchDevelop, along with a discussion of the various tradeoffs we had to make to achieve a reasonable live programming experience. TouchDevelop is public, free to use, and runs in most browsers on any device; we encourage the reader to check it out [1]. We have also produced an 8-minute video that demonstrates the live programming feature.[2]

Much of our work was focused on improving the user experience for the programmer. Our key finding here was to place the live and code view side by side while making elements in each view navigable to elements in the other view. The live view is automatically scaled down to fit on a smaller portion of the screen, but we support interactive zooming to allow programmers to inspect the effect of detail adjustments (such as margins and font sizes). Also, because nested boxes often cover their containers completely, we support a nested selection mode where the user can tap the same box multiple times to select enclosing boxes.

One limitation of our system is the representation of a UI program's model as a collection of global variables, where the view itself cannot retain any state. For example, the value of a slider widget must be defined as a global variable, which is then passed into render code to be read and manipulated. Our strict separation of model and view thus conflicts with the encapsulation principle. How to support encapsulation of state in view elements, and how to deal with tricky initialization semantics, remain to be addressed by future work.

Live programming can be an alternative to step-wise debuggers, given the easy navigability between code and rendered UI artifacts.

[2] `http://bit.ly/itsalive13` or directly `http://youtu.be/XnWgX6cORJM`. Please use HD quality setting.

$$\text{(T-INT)}\ \frac{}{C;\Gamma \vdash_\mu n : \mathsf{number}} \qquad \text{(T-STRING)}\ \frac{}{C;\Gamma \vdash_\mu s : \mathsf{string}}$$

$$\text{(T-VAR)}\ \frac{}{C;\Gamma, x:\tau \vdash_\mu x : \tau} \qquad \text{(T-TUPLE)}\ \frac{C;\Gamma \vdash_\mu e_i : \tau_i}{C;\Gamma \vdash_\mu (e_1..e_n) : (\tau_1..\tau_n)}$$

$$\text{(T-LAM)}\ \frac{C;\Gamma, x:\tau_1 \vdash_{\mu_1} e : \tau_2}{C;\Gamma \vdash_{\mu_2} \lambda(x:\tau).e : \tau_1 \xrightarrow{\mu_1} \tau_2} \qquad \text{(T-SUB)}\ \frac{C;\Gamma \vdash_\mu e : \tau_1 \xrightarrow{\mathsf{p}} \tau_2}{C;\Gamma \vdash_\mu e : \tau_1 \xrightarrow{\mu_2} \tau_2}$$

$$\text{(T-APP)}\ \frac{C;\Gamma \vdash_\mu e_1 : \tau_1 \xrightarrow{\mu} \tau_2 \quad C;\Gamma \vdash_\mu e_2 : \tau_1}{C;\Gamma \vdash_\mu e_1 e_2 : \tau_2} \qquad \text{(T-FUN)}\ \frac{\mathsf{fun}\ f : \tau_1 \xrightarrow{\mu_2} \tau_2\ \mathsf{is}\ e_2 \in C}{C;\Gamma \vdash_\mu f : \tau_1 \xrightarrow{\mu_2} \tau_2}$$

$$\text{(T-BOXED)}\ \frac{C;\Gamma \vdash_{\mathsf{r}} e : \tau}{C;\Gamma \vdash_{\mathsf{r}} \mathsf{boxed}\ e : \tau} \qquad \text{(T-POST)}\ \frac{C;\Gamma \vdash_{\mathsf{r}} e : \tau}{C;\Gamma \vdash_{\mathsf{r}} \mathsf{post}\ e : ()}$$

$$\text{(T-ATTR)}\ \frac{\Gamma_{\mathsf{a}}(a) = \tau \quad C;\Gamma \vdash_{\mathsf{r}} e : \tau}{C;\Gamma \vdash_{\mathsf{r}} \mathsf{box}.a := e : ()}$$

$$\text{(T-GLOBAL)}\ \frac{\mathsf{global}\ g : \tau = v \in C}{C;\Gamma \vdash_\mu g : \tau} \qquad \text{(T-ASSIGN)}\ \frac{\mathsf{global}\ g : \tau = \in C \quad C;\Gamma \vdash_{\mathsf{s}} e : \tau}{C;\Gamma \vdash_{\mathsf{s}} g := e : ()}$$

$$\text{(T-PUSH)}\ \frac{C(p) = (e_1, e_2) \quad C;\Gamma \vdash_{\mathsf{s}} e : \tau}{C;\Gamma \vdash_{\mathsf{s}} \mathsf{push}\ p\ e : ()} \qquad \text{(T-POP)}\ \frac{}{C;\Gamma \vdash_{\mathsf{s}} \mathsf{pop} : ()}$$

$$\text{(T-PROJ)}\ \frac{C;\Gamma \vdash_\mu e : (\tau_1..\tau_n)}{C;\Gamma \vdash_\mu e.i : \tau_i}$$

Figure 10. Expression type rules.

$$\text{(T-SYS)}\ \frac{\begin{array}{c} C \vdash C \quad C \vdash D \quad C \vdash S \\ C \vdash P \qquad C \vdash Q \\ \mathsf{page\ start}().. \in C \end{array}}{\vdash (C, D, S, P, Q)} \qquad \text{(T-EMPTY)}\ \frac{}{C \vdash \epsilon}$$

$$\text{(T-D-INV)}\ \frac{}{C \vdash \bot} \qquad \text{(T-B-VAL)}\ \frac{C \vdash B}{C \vdash B\ v}$$

$$\text{(T-B-ATTR)}\ \frac{C \vdash B \quad \Gamma_{\mathsf{a}}(a) = \tau \quad C;\epsilon \vdash_{\mathsf{p}} v : \tau}{C \vdash B\ [a = v]} \qquad \text{(T-B-NEST)}\ \frac{C \vdash B_i}{C \vdash B_1\ \langle B_2 \rangle}$$

$$\text{(T-C-GLOBAL)}\ \frac{\begin{array}{c} C \vdash C' \quad g \notin \mathit{Defs}(C') \\ \tau\ \text{is}\ \rightarrow\text{-free} \\ C;\epsilon \vdash_{\mathsf{p}} v : \tau \end{array}}{C \vdash C'\ \mathsf{global}\ g : \tau = v} \qquad \text{(T-C-FUN)}\ \frac{\begin{array}{c} C \vdash C' \quad f \notin \mathit{Defs}(C') \\ C;\epsilon \vdash_{\mathsf{p}} e : \tau_1 \xrightarrow{\mu} \tau_2 \end{array}}{C \vdash C'\ \mathsf{fun}\ f : \tau_1 \xrightarrow{\mu} \tau_2\ \mathsf{is}\ e}$$

$$\text{(T-C-PAGE)}\ \frac{\begin{array}{c} C \vdash C' \quad p \notin \mathit{Defs}(C') \\ \tau\ \text{is}\ \rightarrow\text{-free} \\ C;\epsilon \vdash_{\mathsf{s}} e_1 : \tau \xrightarrow{\mathsf{s}} () \\ C;\epsilon \vdash_{\mathsf{s}} e_2 : \tau \xrightarrow{\mathsf{r}} () \end{array}}{C \vdash C'\ \mathsf{page}\ p(\tau)\ \mathsf{init}\ e_1\ \mathsf{render}\ e_2}$$

$$\text{(T-S-ENTRY)}\ \frac{C \vdash S \quad C;\epsilon \vdash_{\mathsf{p}} v : \tau}{C \vdash S\ [g \mapsto v]} \qquad \text{(T-R-ENTRY)}\ \frac{\begin{array}{c} C \vdash P \qquad C;\epsilon \vdash_{\mathsf{p}} v : \tau \\ \mathsf{page}\ p(\tau)\ \mathsf{init}\ e_1\ \mathsf{render}\ e_2 \in C \end{array}}{C \vdash P\ (p, v)}$$

$$\text{(T-Q-EXEC)}\ \frac{C \vdash Q \quad C;\epsilon \vdash_{\mathsf{p}} v : () \xrightarrow{\mathsf{s}} ()}{C \vdash Q\ [\mathsf{exec}\ v]} \qquad \text{(T-Q-PUSH)}\ \frac{\begin{array}{c} C \vdash Q \qquad C;\epsilon \vdash_{\mathsf{p}} v : \tau \\ \mathsf{page}\ p(\tau)\ \mathsf{init}\ e_1\ \mathsf{render}\ e_2 \in C \end{array}}{C \vdash Q\ [\mathsf{push}\ p\ v]}$$

$$\text{(T-Q-POP)}\ \frac{C \vdash Q}{C \vdash Q\ [\mathsf{pop}]}$$

Figure 11. System state type rules.

However, not all aspects of program execution may be sufficently visible in the view. Also, the code in event handlers and initialization bodies is not debuggable via live programming. Thus, a step-wise debugger is still useful and future work may look at how live programming and step-wise debugging can work together. Alternatively, we may explore enhancing the programming model so that even state-changing code can be debugged through live programming, as in the Subtext language [7]. One avenue to explore is the use of boxed statements to produce debugging output in batch computations.

Our model re-executes the **render** code of the current page being viewed, whenever this render code or the program's model changes. Recreating the entire box tree on a redraw can become slow if there are many boxes on the screen. We are currently working on a simple optimization where we can reuse box tree elements that have not changed. An intriguing avenue for future work is the application of research on self-adjusting computation [2], which would allow redundant parts of the render computation to be elided automatically.

6. Related Work

As mentioned previously, visual programming languages have long supported live programming. Burnett *et al.* [5] survey these languages and detail how they can support live programming efficiently. In contrast, we describe how live programming can be supported in the context of a fully expressive textual, imperative language with standard control flow constructs.

Superglue [17] is a textual language that adopts a model inspired by dataflow visual languages; live programming is indeed achieved, but many programs are hard to express with dataflow alone.

Flogo II [11] is another textual live language that supports *live text*, where the state of an executing program is presented as graphical annotations in the code. Rather than annotating code with execution details, our work focuses on making the connection navigable between code and the program's execution (as the rendered UI).

Live programming depends on a "model" that persists between program edits, which originates from Smalltalk's support for image-based persistence [9]. Although Smalltalk supports "fix-and-continue," it does not provide live programming as code edits and execution are independent. Self [26] with its Morphic [16] UI library gets around this limitation by supporting the direct manipulation [24] of object run-time structures, although such edits only persist and do not affect the object's code. In contrast, our work is able to support liveness through a deep connection between code and program execution as well as direct manipulation whose effects are enshrined in code.

Going beyond live programming, Subtext [7] explores how code and program execution can be represented using the same encoding; code in Subtext is not so much executed as it is copied. Our goal is less ambitious: we view live programming as a promising next step in bridging the gap between code and program execution.

Mobl [13] is a programming language for mobile devices. It provides page stack navigation and view-model separation (using data binding), but no live programming. HyperCard [4] provides an overall experience that is quite similar to ours (page stack navigation, persistent state, event handling, and quick switching between editing the code and interacting with the program). However, programming is not live, as code does not automatically reexecute, nor is there any support for writing specialized code to express the relationship between view and state.

Live programming is also related to the hot swapping of code, introduced by Fabry [8], an important capability of the Erlang language [3], both of which allow the code of an executing program to be updated without losing its state and context. However, live programming is concerned with program development while code hot swapping is concerned with updating programs already deployed. The former focuses more on navigable connections between code and execution, while the latter focuses more on uptime and robustness.

Hicks et al. [12] propose live software updating via state snapshots, state transformations, and re-starting of the changed program. The reconstruction of the call-stack and application of data transformation is the responsibility of the programmer, rather than an automated system. This provides maximum flexibility, while making it unsuitable for live programming.

7. Conclusion

Live programming is an idea whose time has come: emerging interactive programming systems [10, 22, 27] capture the imagination of today's programmers and promise to narrow the temporal and perceptive gap between program development and code execution. This paper has shown how live programming can be conceptualized and realized in an expressive procedural language, by tightly integrating UI construction techniques (model-view separation and page-stack navigation) with programming language techniques (syntactically nested boxes and a type and effect system). By providing a formal model, we have established a foundation for critical discussions at a technical level. Future work on live programming may explore improvements in expressiveness, such as support for state encapsulation in the view, or improvements in performance, such as optimizations that help to scale to larger and more complex user interfaces.

References

[1] TouchDevelop website and web application (Microsoft Research). `http://www.touchdevelop.com`.

[2] U. A. Acar. Self-adjusting computation: (an overview). In *Proc. of Partial Evaluation and Program Manipulation (PEPM)*, pages 1–6, 2009.

[3] J. L. Armstrong and S. R. Virding. Erlang - an experimental telephony programming language. In *Proc. of International Switching Symposium*, pages 2–7, 1990.

[4] B. Atkinson. Hypercard. Apple Computer, 1987.

[5] M. M. Burnett, J. W. Atwood Jr, and Z. T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *Proc. of the IEEE Symposium on Visual Languages*, pages 126–134, 1998.

[6] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, P. J. V. Zee, and S. Yang. The scaling-up problem for visual programming languages. Technical report, Oregon State University, 1994.

[7] J. Edwards. Subtext: uncovering the simplicity of programming. In *Proc. of OOPSLA Onward!*, pages 505–518, 2005.

[8] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proc. of ICSE*, pages 470–476, 1976.

[9] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[10] C. Granger. Light Table - a reactive work surface for programming. `http://www.kickstarter.com/projects/ibdknox/-light-table`, 2012.

[11] C. M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003. AAI0805688.

[12] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster. Kitsune: efficient, general-purpose dynamic software updating for C. In *Proc. of OOPSLA*, Oct. 2012.

[13] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In *Object oriented programming systems languages and applications (OOPSLA)*, pages 695–712. ACM, 2011.

[14] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *Proc. of CHI*, pages 480–486, 1995.

[15] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *In Principles of Programming Languages (POPL*, pages 47–57. ACM Press, 1988.

[16] J. H. Maloney and R. B. Smith. Directness and liveness in the Morphic user interface construction environment. In *Proc. of UIST*, pages 21–28, nov 1995.

[17] S. McDirmid. Living it up with a live programming language. In *Proc. of OOPSLA Onward!*, pages 623–638, October 2007.

[18] C. Muratori. Immediate-mode graphical user interfaces. `www.molly-rocket.com/861`, 2005.

[19] D. A. Norman and S. W. Draper. *User Centered System Design; New Perspectives on Human-Computer Interaction*. L. Erlbaum Associates Inc., 1986.

[20] M. Puckette. Pure Data: another integrated computer music environment. In *Proc. of International Computer Music Conference*, pages 37–41, 1996.

[21] T. Reenskaug. Thing-model-view-editor an example from a planningsystem, `http://heim.ifi.uio.no/~trygver/1979`. Technical report, Xerox PARC, 1979.

[22] J. Resig. Khan Academy - computer science. `http://www.khan-academy.org/cs`.

[23] E. Sandewall. Programming in an interactive environment: the "LISP" experience. *ACM Computing Surveys*, 10(1):35–71, Mar. 1978.

[24] B. Shneiderman. Direct manipulation. a step beyond programming languages. *IEEE Transactions on Computers*, 16(8):57–69, August 1983.

[25] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. TouchDevelop - programming cloud-connected mobile devices via touchscreen. In *Proc. of SPLASH Onward!*, 2011.

[26] D. Ungar and R. B. Smith. Self: the power of simplicity. In *Proc. of OOPSLA*, pages 227–242, December 1987.

[27] B. Victor. Inventing on principle. Invited talk at the Canadian University Software Engineering Conference (CUSEC), Jan. 2012.