INCREMENTAL FUNCTIONAL REACTIVE PROGRAMMING FOR INTERACTIVE MUSIC SIGNAL PROCESSING

Caleb Reach

Dept. of Computer Science
Virginia Tech
Blacksburg, Virginia
`jtxx000@cs.vt.edu`

ABSTRACT

Textual music programming languages offer greater expressive power than diagrammatic visual programming languages and semi-modular graphical user interfaces. However, textual music programming languages don't allow fine-grained incremental updates to the signal flow graph—instead, they only allow course-grained updates at the statement level. In both the diagrammatic visual programming language and the graphical user interface paradigms, users can directly adjust a parameter by editing the value inline, and such an adjustment does not affect the state of any other part of the signal flow graph. For example, adjusting the attack time of an envelope does not affect the contents of a delay line. By contrast, users of textual music programming languages must either: 1) assign names to nodes and then later use a separate statement to adjust the parameter value, or 2) lose node states (eg. envelope positions, instantaneous LFO phases, and delay line contents) by reevaluating the entire original statement or program. We present a new paradigm in which users can directly edit programs without losing state. In our approach, time-varying programs evaluate to time-varying signal processing graphs, and incremental updates to the program result in incremental updates to the signal processing graph.

1. INTRODUCTION

Ideally, platforms for creative music signal processing should exhibit two characteristics:

Incrementality Music signal processing is inherently exploratory. Often, users do not have an exact sound in mind when they use synthesis software; rather, they start with a simple configuration (eg. a simple saw wave) and iteratively modify the sound until they find something appealing. To support rapid iteration, music software must allow users to incrementally modify parameters in real-time while preserving the state residing in the components of the signal flow graph. For example, suppose a user is controlling a basic subtractive synthesizer with a MIDI controller. The user should be able play notes with the MIDI controller while simultaneously adjusting the cutoff frequency of a low-pass filter without losing the states of LFOs and envelopes (that is, the phases of the LFOs should not be reset and the positions of the envelopes should not be reset).

Flexibility Music signal processing software should not impose artificial limitations on users. For example, software synthesizers should not artificially limit users to n oscillators or k envelopes; such limitations are inherently necessary in analog synthesizers, but are artificial in music software.

The three prevalent paradigms for interactive computer-based music signal processing make different design trade-offs in terms of these characteristics:

Simple GUI Many software synthesizers offer a virtual recreation of a hardware synthesizer interface. These interfaces are easy to learn and use but offer limited flexibility in routing.

Diagrammatic Visual Programming Language Another common paradigm is to recreate the module-and-patch-cord interfaces of modular synthesizers. Diagrammatic visual programming languages offer significantly more expressive power than simple GUIs, but they can be harder to learn. Examples include PureData [11], Max/MSP, Reaktor, and SynthMaker.

Imperative Textual Programming Language Most music programming languages suitable for interactive use rely on imperative updates for incrementality. For example, a user might instantiate a filter node and assign it to the variable x. The user may then change the cutoff frequency by assigning a new frequency to the cutoff frequency slot with a statement such as $x.f \leftarrow 440$. Examples of this paradigm include SuperCollider [9] and ChucK [14].

We present a new paradigm which we call *incremental functional reactive programming*. In this paradigm, users incrementally modify a functional program that evaluates to a signal flow graph. Incremental changes to the signal flow graph are implicitly derived from the user's incremental changes to the program. Our approach has two advantages over imperative music programming languages:

1. At any given point in time, the complete declarative specification of the program (modulo node state) is available to the user. In contrast, with an imperative language and REPL, users must read through the entire interaction history to properly deduce the signal flow graph.
2. With our approach, users do not need to assign names to nodes in order to modify them in the future. For example, a user may specify

```
(lp-filter (+ 1000
              (* (sin-lfo 1)
                 220))
           0.8
           audio)
```

to produce a lowpass filter with the cutoff frequency modulated by a sinusoidal LFO. The user can then modify the code to

```
(lp-filter (+ 1000
              (* (sin-lfo 2)
                 220))
           0.8
           audio)
```

to change the rate of the LFO without changing its phase. With an imperative language, this would not be possible without giving the LFO a name.

2. INCREMENTAL FUNCTIONAL REACTIVE PROGRAMMING

We now describe our approach in depth.

2.1. State mapping

As each signal processing node may be stateful, the nodes at any given revision must be matched with corresponding nodes from the previous revision in order to carry over the node states from the previous revision. For example, consider the following program:

```
(define (f hz)
  (* (sin-lfo hz)
     (sin-lfo (* 2 hz))))

(define x (+ (f 2) (f 3)))
```

Here we have four sinusoidal LFOs: two for each invocation of the function `f`. Let $\phi_1^{(k)}(t), \phi_2^{(k)}(t), \ldots, \phi_n^{(k)}(t)$ denote the instantaneous phase of each LFO at the k^{th} revision of the program. Furthermore, let these be ordered by the order in which they would be evaluated, assuming a call-by-value evaluation strategy with left-to-right argument evaluation order. Suppose that user enters the initial revision of the program T_0 at time t_0 and then, at time t_1, changes the program to T_1:

```
(define (f hz)
  (* (sin-lfo hz)
     (sin-lfo (* 2 hz))))

(define x (+ (f 2) (f 3) 2))
```

Suppose that $\phi_i^{(0)}(t_0) = 0$ for all $i \in 1..4$—that is, each LFO is initialized with an instantaneous phase of zero. In order to satisfy the property of incrementality, we wish for $\phi_i^{(0)}(t_1) = \phi_i^{(1)}(t_1)$.

Now suppose the user changes the program to T_2 at time t_2

```
(define (f hz)
  (* (sin-lfo hz)
     (sin-lfo (* 2 hz))))

(define x (+ (f 5) (f 2) (f 3) 2))
```

We now have six invocations of `sin-lfo`. How should we define the mapping between $\phi_i^{(1)}(t_2)$ and $\phi_j^{(2)}(t_2)$ for $j \in 1..6$? The simplest solution would perhaps be to have

$$\phi_j^{(2)}(t_2) = \begin{cases} \phi_j^{(1)}(t_2) & j \in 1..4 \\ 0 & j \in 5..6 \end{cases}$$

However, this replaces the phases of the LFOs invoked through the call `(f 3)` with zero when it would be more parsimonious if the state for the call to `(f 3)` was carried over. The problem is worse with larger programs containing many modules of the same type. With this mapping strategy, any newly inserted module $f_i^{(k)}$ is initialized with the state of some existing module $f_i^{(k-1)}$ and all subsequent modules are given states that may differ wildly from their previous states.

We argue that this mapping should depend on how the user edited the program source code. If the user inserts a node term, the resulting signal flow node should be initialized with a default state, and all other nodes should keep their current state. If the user inserts a function invocation, all nodes associated with this new function invocation should be initialized with a default state, and all other nodes should keep their state. In the example above, we should have the mapping

$$\phi_j^{(2)}(t_2) = \begin{cases} 0 & j \in 1..2 \\ \phi_j^{(1)}(t_2) & j \in 3..6 \end{cases}$$

because the new function invocation is inserted before the other invocations.

If the user duplicates a node term, both of the resulting signal flow nodes should be initialized with the state of the original node term. For example, if the user goes from

```
(+ 1 (sin-lfo 2))
```

to

```
(+ 1 (sin-lfo 2) (sin-lfo 3))
```

by copying and pasting `(sin-lfo 2)` and then modifying the argument, then the new LFO should start with the same phase as the copied LFO. If, on the other hand, the user enters `(sin-lfo 3)` by hand, the LFO should start with a default phase.

If the user moves a node term into the body of a function, then the node's state is duplicated for each invocation of the function. For example, if the user changes the program from

```
(define (f x)
  (* x 2))

(+ (f 2) (f 3) (sin-lfo 2))
```

to

```
(define (f x)
  (* x 2 (sin-lfo x)))

(+ (f 2) (f 3))
```

by cutting and pasting the `sin-lfo` term inside the function, then both LFOs resulting from the call to `f` should begin with the same state as the original LFO from the previous revision.

If the user moves a node term outside the body of a function, the node's state must be initialized with a default value, as there is no way to determine from which function invocation the node's state should be derived. For example, if the previous edits were done in reverse, the two LFOs would likely be out of phase, and so there would be no easy way to choose which phase to keep.

2.2. Description

To keep track of whether terms were added, copied, modified, or deleted between revisions, we associate a label with each term in the program. All labels are unique in a given revision, but labels may be reused across revisions. To track labels across revisions, we use an *updated label* of the form $a \rightsquigarrow b$. There are three types of terms which use such labels: λ terms $(\lambda v.T)^{a \rightsquigarrow b}$, application terms $(T_1\ T_2)^{a \rightsquigarrow b}$, and node terms[1] $(n_i\ T)^{a \rightsquigarrow b}$. The updated label states that the term had label a in the previous revision and now has b in the current revision. By allowing labels to change with each revision, we permit node copying, where a single node in revision k becomes two nodes in revision $k+1$.

2.3. Evaluation

Interpreters for applicative programming languages commonly use environments to keep track of variable mappings. When a λ function is evaluated, a closure is created that captures the variable bindings in the current environment. With our programming language, we maintain both a variable map and a *context*, which stores a mapping between labels and *instances* of nodes, functions, and applications. We say that a term is *instantiated* when it is first evaluated and *matched* when it is evaluated in a future revision in such a way that state from the previous revision should be retained.

We define a big-step operational semantics operator $T|\gamma, \beta \Longrightarrow V|\beta'$ that maps from terms with the given context β and variable map γ to values with a new context β'. We deviate slightly from traditional presentations by using closures: in our presentation, λ terms evaluate to λ values that capture both the variable map and context from the current environment. To evaluate some term $(\lambda v.T)^{a \rightsquigarrow b}$, we first look for a λ instance that matches the previous label:

$$\frac{L = \beta(a)}{(\lambda v.T)^{a \rightsquigarrow b}|\gamma, \beta \Longrightarrow (\lambda v.T)^{\beta(a) \rightsquigarrow \varphi}_{\gamma,\beta}|b \mapsto \varphi}$$

where $L = \beta(a)$ iff L is a member of the set of λ instance labels and the context β maps the label a to L, and $\varphi \in L$ is a λ instance label unique to this reduction[2]. The set of λ instance labels is used to give each λ value a unique identity and to permit comparison between λ values. The notation $(\lambda v.T)^{\delta \rightsquigarrow \rho}_{\gamma,\beta}$ denotes a λ value with the updated λ instance label $\delta \rightsquigarrow \rho$. The notation $b \mapsto \varphi$ denotes a context in which the label b maps to the instance label φ. If no λ instance can be found, we instantiate the λ:

$$\frac{L \neq \beta(a)}{(\lambda v.T)^{a \rightsquigarrow b}|\gamma, \beta \Longrightarrow (\lambda v.T)^{\varepsilon \rightsquigarrow \varphi}_{\gamma,\beta}|b \mapsto \varphi}$$

where ε denotes the null label and $L \neq \beta(a)$ iff either $a \notin \beta$ or $\beta(a)$ is not a λ instance label.

[1]For simplicity, our description of the language here only describes semantics for unary functions and nodes, though the description can be generalized to n-ary functions without difficulty. Our description is no less general, as currying and church-encoded tuples can be used to represent n-ary functions.

[2]It is not difficult to modify the semantics in order to thread a sequence of unique labels through the reductions explicitly. In the interests of preventing an already-dense notation from becoming impenetrable, we elide this formality.

To reduce an application, we first try to find an application instance.

$$\frac{\begin{array}{c} T_1|\gamma, \beta \Longrightarrow (\lambda v.T)^{\delta \rightsquigarrow \rho}_{\hat{\gamma},\hat{\beta}}|\beta'_1 \\ A^\delta = \beta(a) \\ T_2|\gamma, \beta \Longrightarrow V_1|\beta'_2 \\ T|v \mapsto V \cup \hat{\gamma}, A^\delta \cup \hat{\beta} \Longrightarrow V_2|\dot{\beta} \end{array}}{(T_1\ T_2)^{a \rightsquigarrow b}|\gamma, \beta \Longrightarrow V_2|b \mapsto \dot{\beta}^\rho}$$

where A^δ denotes a member of the set of application instances with the λ instance label δ and $\gamma_1 \cup \gamma_2$ denotes a combination of two variable maps with preference given to the left map. The notation $\beta_1 \cup \beta_2$ works similarly. The λ instance label comes from the function that T_1 evaluates to. If there is no application instance, we instantiate the application:

$$\frac{\begin{array}{c} T_1|\gamma, \beta \Longrightarrow (\lambda v.T)^{\delta \rightsquigarrow \rho}_{\hat{\gamma},\hat{\beta}}|\beta'_1 \\ A^\delta \neq \beta(a) \\ T_2|\gamma, \beta \Longrightarrow V_1|\beta'_2 \\ T|v \mapsto V \cup \hat{\gamma}, \hat{\beta} \Longrightarrow V_2|\dot{\beta} \end{array}}{(T_1\ T_2)^{a \rightsquigarrow b}|\gamma, \beta \Longrightarrow V_2|b \mapsto \dot{\beta}^\rho}$$

A variable reference v is simply looked up in the variable map γ:

$$\frac{}{v|\gamma, \beta \Longrightarrow \gamma(v)|\varnothing}$$

Matched nodes are evaluated by

$$\frac{N_i = \beta(a) \qquad T|\gamma, \beta \Longrightarrow V|\beta'}{(n_i\ T)^{a \rightsquigarrow b}|\gamma, \beta \Longrightarrow n_i^{N_i \rightsquigarrow \varphi} V|b \mapsto \varphi}$$

where N_i denotes a member of the set of node instance labels for node type i and φ is a node instance label that is unique to the evaluation of this term. Nodes that must be instantiated are evaluated with

$$\frac{N_i \neq \beta(a) \qquad T|\gamma, \beta \Longrightarrow V|\beta'}{(n_i\ T)^{a \rightsquigarrow b}|\gamma, \beta \Longrightarrow n_i^{N_i \rightsquigarrow \varphi} V|b \mapsto \varphi}$$

Note that the sets of λ instance labels, application instances, and each set of node instance labels are all disjoint.

Let bindings bind a variable without introducing a new context:

$$\frac{T_1|\beta, \gamma \Longrightarrow V_1|\beta'_1 \qquad T_2|\beta, v \mapsto V_1 \cup \gamma \Longrightarrow V_2|\beta'_2}{(v \leftarrow T_1; T_2)|\beta, \gamma \Longrightarrow V_2|\beta'_1 \cup \beta'_2}$$

3. IMPLEMENTATION

Our implementation combines an Emacs [13] extension with a programming language implementation written in Haskell [7]. Before sending the current revision to the Haskell process, the Emacs extension visits each pair of balanced parenthesis. The sexp is transformed from $(s_1\ s_2\ \ldots\ s_n)$ to $((a\ .\ b)\ s_1\ s_2\ \ldots\ s_n)$ and then sent to the Haskell process, where a is label associated with the opening parenthesis and b is a new label generated using `gensym`. After the sexp is sent, the opening parenthesis is assigned the label b. Labels are assigned to characters using Emacs's text properties feature so that when text is copied or moved, the labels are also copied or moved. We use `paredit` [1] to prevent users from accidentally deleting and reinserting parenthesis.

Sexps are then translated straightforwardly into terms in the language. Definitions of the form

```
(define x expr)
rest
```

translate to $x \leftarrow T_1; T_2$, where T_1 is the translation of `expr` and T_2 is the translation of `rest`. Definitions of the form

```
(define (f x) expr)
```

are translated into

```
(define f (lambda (x) expr))
```

Applications, lambda expressions, and variable references are translated as one would expect.

Our current implementation outputs a simple specification of the signal graph updates and is not tied to any specific synthesis engine. We plan to integrate our code with the SuperCollider engine, although this may not allow for node copying.

4. RELATED WORK

Object oriented languages like Smalltalk [6] allow incremental modification of programs. However, like other imperative languages, these modifications are course-grained. Classes in an object oriented language are somewhat analogous to λ expressions in incremental functional reactive languages, and objects are somewhat analogous to λ instances. Modifying a class or λ expression changes all instances of that class or λ expression. However, λ expressions are much more lightweight than classes in an object oriented language.

Faust [10] is a functional programming language for music signal processing. Like our approach, Faust generates a signal flow graph from a functional specification of the graph. Unlike our approach, Faust is not intended for interactive use. The design of a compiled incremental Faust-like language would be interested research area.

In the functional reactive programming paradigm [5], programmers manipulate signals—time-varying values. For example, the position of a modulation wheel can be thought of as a number that changes over time. The state of a midi keyboard key can be seen as a boolean that changes over time representing whether or not the key is currently pressed.

Subtext [3] provides a unique programming paradigm where programs are edited live while running. However, it does not support state preservation as described here. Tangible functional programming [4] provides a framework that seeks to address some of the problems described in this paper, but it deviates from standard programming language syntax.

Scratch [12] is a programming environment with a focus on pedagogy. Users connect together tiles representing statements and expressions to create a program. Alice [2] is a similar language, also focused on pedagogy. The editor concepts from these languages are applicable to the incremental functional reactive programming.

Barista [8] can create structure editors that can track the identity of terms over time. Future research could combine techniques from Barista with the programming language techniques described in this paper.

5. CONCLUSION AND FUTURE WORK

We have presented *incremental functional reactive programming*, a paradigm in which time-varying programs evaluate to time-varying signal processing graphs. Our approach allows users to declaratively specify the signal flow graph while preserving state as the program is edited.

There are several directions in which we would like to continue our research. Firstly, we would like to look into alternative editors for programs. Because our semantics allow for rapid incremental changes to the program, we would like to research the embedding of widgets into the program source. For example, an editor might allow a knob to be placed inline into an expression, allowing the user to drag the knob and hear the results in real-time. Output widgets could also be placed inline—for example, a user might insert an meter inline to analyze at the peak amplitude at a specific point in the signal flow graph.

We would also like to explore the pedagogical possibilities of this work. Scratch and Alice both offer an exploratory environment in which to learn to program imperatively, where users can experiment with program modifications and see the results instantly. An analogous environment for functional programming could be developed using the techniques described in this paper.

In the language described here, terms evaluate to a signal flow graph. Though past computation can affect current computation within the signal flow graph, there is no way for past computation to affect term evaluation within the programming language. Adding a fold primitive to fold over successive values for a term would allow past computation to affect current computation. Such a primitive could be used for reactive programming in circumstances that do not require hard real-time performance.

Our implementation is interpreted rather than compiled. Future work could research the design of compilers for incremental functional programming languages.

Overall, we think that the design of incremental functional reactive programming languages for interactive music signal processing may prove an interesting area for further research.

6. REFERENCES

[1] Taylor R. Campbell. `paredit.el`—minor mode for editing parentheses, July 2005.

[2] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000.

[3] Jonathan Edwards. Subtext: uncovering the simplicity of programming. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 505–518, New York, NY, USA, 2005. ACM.

[4] Conal M. Elliott. Tangible functional programming. *SIGPLAN Not.*, 42:59–70, October 2007.

[5] Conal M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.

[6] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., 1983.

[7] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on

the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[8] A.J. Ko and B.A. Myers. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 387–396. ACM, 2006.

[9] J. McCartney. Supercollider, a new real time synthesis language. In *Proceedings of the International Computer Music Conference*, pages 257–258. INTERNATIONAL COMPUTER MUSIC ACCOCIATION, 1996.

[10] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of faust. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 8(9):623–632, 2004.

[11] M. Puckette. Pure data: another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[12] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[13] Richard M Stallman. *EMACS the extensible, customizable self-documenting display editor*, volume 2. ACM, 1981.

[14] G. Wang. *The ChucK audio programming language." A strongly-timed and on-the-fly environ/mentality"*. PhD thesis, Princeton University, 2009.