Script-n-Scribe: Prodirect Manipulation for Music Composition

Teddy Sudol

CS 691 NN Fall 2015

Abstract

Prodirect manipulation is a user interaction paradigm that combines direct manipulation (e.g. drawing a shape) with programmatic manipulation (e.g. describing a shape in code) by using program synthesis to ensure the program and the directly-manipulated output are synchronized. This idea can be applied to any domain that has both direct manipulation interfaces and programmatic manipulation interfaces, such as music composition.

This paper presents Script-n-Scribe, a prodirect manipulation system for music composition. Combining a small music DSL with a direct manipulation interface, the system uses trace-based synthesis to synchronize the program with the output. It extends the work of Chugh, Albers, and Spradlin, "Program Synthesis for Direct Manipulation Interfaces" (2015) to the domain of music, along with augmenting synthesis to use user actions to drive synthesis of updates that alter program structure.

1. Introduction

In their paper *Program Synthesis for Direct Manipulation Interfaces* [2], Chugh et al. develop the idea of prodirect manipulation, the fusion of direct manipulation and programmatic manipulation into one interaction mode. Direct manipulation interfaces allow users to interact with their data, creating an intuitive user experience. On the other hand, programmatic manipulation systems use programming languages to define their output, giving users the full power of programming abstractions. Microsoft Word versus LaTeX for document creation is one example of this dichotomy, which can be found in diverse domains as spreadsheets, vector graphics, presentations and Web app design.

The key idea of prodirect manipulation is to make the interactions "bidirectional." When the program is changed, the output changes; similarly, when the output is changed by the user, the program is updated to reflect the change. The user can comfortably work with both the direct interface and the programming interface, using the strongest features of both. The authors showed the validity of this idea by developing `sketch-n-sketch`, a prodirect manipulation system for creating SVG images. In order to synchronize the program after the user changes the output, they implemented trace-based synthesis, a program synthesis algorithm that records the locations and calculations related to numeric constants in the original program. It uses these traces to calculate the updates needed to transform the original program to match the new output.

The goal of this project is to create a prodirect manipulation system for music composition and adapt trace-based synthesis to a music DSL. Digital music has a history of both direct and programmatic interfaces, lending it well to the idea of prodirect manipulation. For direct manipulation, the user can place notes on a score, while the programmatic side uses a language that can be used to reason about musical primitives.

The remainder of this paper discusses the design of Script-n-Scribe, the prodirect manipulation system for music composition (section 3), including discussions about trace-based synthesis (section 4) and implementation details (section 5).

1.1 Contributions

1. Breve, a small Haskell-inspired DSL for functional music composition,
2. Script-n-Scribe, a prodirect manipulation system that combines Breve and a direct manipulation UI,
3. An adaption of trace-based synthesis for composing music.

2. Background and Related Work

2.1 Sketch-n-Sketch

This project is directly inspired by `sketch-n-sketch`, the prodirect manipulation system created by Chugh et al. [2]. `sketch-n-sketch` includes `little`, a small Lisp-like DSL, as the programming interface for creating SVGs. Beyond the usual base values (numbers, strings, booleans), `little` represents SVG objects as three-element lists, which act as wrappers around the SVG format. When evaluated, these lists create SVG objects that can be rendered in an HTML5 canvas. The user may then modify the SVGs directly by clicking and dragging on control points. `sketch-n-sketch` uses trace-based synthesis to calculate changes to the source program that would make it produce the user's modified output.

2.2 Trace-based Synthesis

Traces are records that tie a numeric value to the program source code. There are two types:

Source code locations: These are very simple traces that record the location of the literal value in the source code or AST. The parser inserts these traces when it encounters a numeric literal.

Expression trace: Expression traces are created during the evaluation of the program. Each primitive operation (addition, subtraction, etc.) combines the traces of its operands into an expression trace. They are essentially miniature ASTs that capture the operations used to calculate the expression value. For example, given the numeric values x located at l_x and y located at l_y, the expression $x + y$ will produce the expression trace $l_x +_t l_y$. Note that this is still a trace, not the actual addition of the two locations.

The trace-based synthesis algorithm is fairly straightforward. It takes as input a map of locations to values in the initial program (e.g. $(l_x \rightarrow x)$, $(l_y \rightarrow y)$) and a list of new trace equations (e.g. $z = l_x +_t l_y$) provided by the user that describe the relationship between a value and a trace. For each new trace, the system attempts to solve for a new value by inverting the expression traces. For example, if the user gives the new value z for the expression trace $l_x +_t l_y$, the algorithm inverts the expression trace to get $z - x = l_y$ and $z - y = l_x$. Each basic operation that creates expression traces must have an inverse.

More formally, the traces from the original program and the user's updated traces form a system of equations. The equations are made from relating the numeric value to the trace, such as $x = t_x$, where t_x is the trace of x. The user's updated traces are hard constraints, and the old traces are soft constraints. The algorithm can attempt to synthesize *faithful* updates, where the synthesis succeeds if and only if every hard constraint is satisfied. It can also try to create *plausible* updates, where at least one hard constraint must be satisfied. The key algorithmic difference is that faithful updates try to update only one location's value for every user update, while plausible updates can allow multiple equations to be solved for the same location.

Trace-based synthesis allows for two kinds of updates. Local updates are simply updates to existing values, like changing x to $z - y$. Structural updates, on the other hand, change the AST of the program by adding new elements, function calls, and so on.

Finally, the "mode" of `sketch-n-sketch` influences synthesis. In *ad hoc* mode, every program update is presented to the user, allowing them to make the final decision about how the program is updated to match the output. In *live* mode, a rotating heuristic determines the final program update, ensuring that the possible changes are spread out over as many unique locations as possible. Additionally, structural updates are only permitted in *ad hoc* mode.

2.3 Computing and Music

Programmatic manipulation for music can be traced back to at least 1957 and the work of Max Mathews, who developed MUSIC I, the first music programming language [5]. MUSIC I is the first in the family of MUSIC-N languages and inspired dozens of other languages.

Csound [3] is one of the more popular descendants of MUSIC-N. Csound uses a low-level, signal-driven view of music. Users can define signal generators, envelopes and filters for adjusting sound, and provide a score that describes the output signals.

Another well-known descendant of MUSIC-N is Max/MSP [4]. Unlike its ancestors, Max is essentially a visual programming language, in which users connect objects on a digital canvas. The connections describe the flow of data (digital audio) through the program. MSP is a commonly-used extension of Max for real-time digital signal processing. One of the popular uses of Max/MSP is as a live-music performance platform, using MSP and Max's built-in hardware interfacing to control the show. Max is named after Max Mathews.

ChucK [12, 13] presents a "strongly timed" view of music. It has some similarities to Csound, in that it focuses on signals instead of notes, but it implements a more data-flow oriented model similar to Max/MSP. Unlike either system, it is heavily based around the concept of time as a programming construct, allowing the user to advance the time of program execution at their discretion.

On the direct manipulation side, programs like Finale [8], Sibelius [10] and Rosegarden [1] provide a skeuomorphic design inspired by classic written scores. These programs commonly use drag-and-drop interfaces, letting users write notes and rests directly onto the score. High-level music theory concepts are available; for example, the user can annotate the score with the phrase "*crescendo poco a poco*", ("gradually getting louder") and the program will interpret it as a slow increase in dynamic without the user having to specify the amount to increase the volume by or the rate at which it is increased. Another common feature is MIDI interfacing, by which users interact with the program using a MIDI-enabled external device, like an electronic piano keyboard.

3. Script-n-Scribe

The main contribution of my project is Script-n-Scribe, a prodirect manipulation system for composing music. While the SVG objects of `sketch-n-sketch` are defined by 3-element lists of `[kind attributes children]`, where *kind* refers to the type of the SVG node (rectangle, circle, etc.), music "objects" are hierarchical structures composed of basic values. At the bottom level are pitches, octaves and durations, which are combined to form notes (pitch coupled with octave, with a duration) and rests (just a duration). Notes and rests are combined to form musical phrases, which are stitched together to create full songs.

This idea of building music by combining objects should sound familiar to programmers, and unsurprisingly many music DSLs and libraries use this approach. I followed this model when developing Breve, the DSL used in Script-n-Scribe, described in section 3.1. Section 3.2 discusses the Script-n-Scribe UI. The third component of the project, the trace-based synthesis system, is discussed in section 4.

3.1 Breve

It seems that creating a DSL is absolutely necessary when creating a prodirect manipulation system. Trace-based synthesis requires support from both the parser and the evaluator of the target language in order to create the traces used in synthesis. Modifying an existing language to produce traces is an arduous task. A DSL can have the additional advantage of being tuned for use with a particular problem domain, by definition, making it easier to work with for users.

The language used in Script-n-Scribe is called Breve, from the Italian for "short" or "`little`." It is a small, dynamically-typed language providing music objects as first-class citizens. The base values of the language are integers, floating-point numbers, booleans and pitch classes. A "pitch class" is simply a note name, such as C, G♯ or D♭; technically it refers to the set of all notes of that name across all octaves [9]. Notes are created by combining a pitch class, an integer (for the octave) and a number (for the duration): `(D 4 1/4)`. Rests are similar, but just consist of a duration: `(rest 1)`. Notes and rests combined form *snippets*, essentially musical phrases. Besides these musical features, Breve has lists, variables, basic math operators, functions, conditionals and pattern matching.

The musical aspects of Breve are backed by the Euterpea [11] library. Developed as part of the *Haskell School of Music* by Paul Hudak [6], Euterpea is "a domain-specific language, embedded in the functional language Haskell, for computer music development." Breve's syntax is in part inspired by Euterpea. For example, in Euterpea the function call `(d 4 (1/4))` creates a note object representing a quarter note (the third argument, `(1/4)`) with the pitch D_4. Similarly, the Breve expression `(D 4 1/4)` creates the same note.

One key difference is that Euterpea does not support the idea of "snippets." In Breve, a snippet is a list of notes and rests that enforces a temporal ordering of its elements: in the snippet $\{n_1, n_2, \dots\}$, n_i will always be played right before n_{i+1}. Snippets have two defined operators, the sequence combinator `:+:` and the parallel combinator `:=:`. $s_1 :+: s_2$ means s_1 and s_2 will be played in sequence, and $s_1 :=: s_2$ means s_1 and s_2 will be played at the same time. These operators are lifted directly from Euterpea, where they are used to combine general "Music" objects like notes and rests. A snippet is essentially an abstraction of the sequence combinator, though it has a special use for user interaction that is discussed in section 3.2.

Breve programs bear a superficial similarity to Haskell programs. Each Breve program is a sequence of assignments, e.g. $x = expr$. The order of these statements is significant, however. The program

```
x = y;
y = 5;
```

will fail to evaluate because x refers to y, which has not yet been defined. Just like in Haskell, the variable "`main`" is significant and indicates the output of the program. A Breve program may return any value, but it will preferably create a musical phrase.

A complete description of the Breve grammar is provided in the file `docs/breve.md`.

3.2 User Interaction

While Breve can be used as a stand-alone DSL, it was designed as part of the Script-n-Scribe environment. The user interface of Script-n-Scribe provides the direct manipulation aspect of the project, namely writing music with a score-like interface.

The UI is inspired by the `sketch-n-sketch` UI, which has a straightforward layout: a code editor on the left half of the screen, the direct manipulation/output window on the right half, and various controls in the middle. The output of the Breve program on the left is drawn as music on the right. As mentioned previously, the `main` variable determines the output of a Breve program, and the (musical) value of `main` is drawn in the music pane as a score, labeled with "main". The user can set various features, such as the type of synthesis, the UI mode, and so on.

However, this UI model does not give the user enough power. For very simple programs, the user will directly modify notes that are defined in the program as part of `main`, or are defined in other variables. Their changes will be directly reflected in the updated variables. But consider a more complex program:

```
d = (D 4 1/4) -- a quarter note D4
-- the major triad built on D4 (a list)
dfsa = arpeggio([0,4,7], d)
-- the arpeggio as a sequence (snippet)
dl = line(dfsa)
-- the arpeggio as a chord
dc = chord(dfsa)
main = dl :+: (rest 1/4) :+: dc
```

`arpeggio`, `line` and `chord` are all functions included in the Breve Prelude (standard library) that is included with every program. This program creates the triad $[\mathtt{D}_4, \mathtt{Fs}_4, \mathtt{A}_4]$ as a list (`arpeggio`), turns it into a snippet (`line`) and a chord (i.e. the notes in parallel, using `chord`), then combines the line and chord, separated by a quarter note rest, to create `main`.

With the UI model outlined above, all the user would see is the `main` snippet in the music pane. If they want to modify any notes using direct manipulation, they are only able to modify the *output* of the functions in `main`, relying on synthesis to carry the changes back to the function's input. For a relatively simple program this may be sufficient, but for more complex programs, the user's changes will be increasingly divorced from the details of the program. Additionally, the complexity of the traces grows with the number of function calls, and longer traces create more possible substitutions to be synthesized. The user could be faced with dozens of possible changes to the source program for every simple change they make.

Instead of forcing the user to modify the outputs in order to change the inputs, the UI can expose more snippets to the user. Since each snippet is a musical phrase, each one can be rendered on a score in the music pane of the UI. But there may be an arbitrary number of snippets in a program, some of which are buried inside functions, so only a subset of all the snippets in the program should be displayed to the user. The most obvious solution is to display all of the top-level snippets in a program. Recall that the `main` snippet is displayed as a score labeled "main" in the music panel; similarly, each snippet can be tagged with its name in the music panel. We can go one step further and display every note, rest and snippet that is defined at the top-level of the program. Using the example above, the user would see 4 snippets: `d`, `dl`, `dc` and `main`. Since `dfsa` is a list of notes, not a snippet, it is not displayed.

The user now has much tighter control of the program on the direct manipulation side. Instead of modifying a note in `main` and hoping the correct substitution comes through, they can more directly manipulate the parameter that corresponds to what they want to change. There's only one layer of indirection between the user and `dfsa`, and they have access to one of the arguments of the `arpeggio` function call.

4. Trace-Based Synthesis

As explained previously, trace-based synthesis uses numeric and expression traces to calculated new values for numeric constants in the source program. A significant part of this project has been adapting trace-based synthesis to work with Breve. Instead of general "numeric values", which `little` can get away with since it uses only floating-point numbers, Script-n-Scribe must trace pitch classes, integers and floating-point numbers. This does not complicate synthesis, besides making the implementation of operators longer to handle all the cases. However, there is one edge case: in Breve, addition and subtraction with pitch classes is defined only for $p + n$ or $p - n$, where p is a pitch class and n is an integer. When inverting the addition of two numbers, the commutative property of addition simplifies inversion, but inverting addition with a pitch class has to be handled as two separate cases to avoid undefined behavior. Fortunately, subtraction is implemented as $p+(-n)$, which is reflected in the trace and therefore handled by the addition rules. Besides the different base value types and some implementation details, trace-based synthesis is largely the same.

4.1 Ad Hoc or Live?

`sketch-n-sketch` as written allows for either ad hoc or live mode to be used. However, in practice it seems that live mode is used exclusively. This makes sense for the SVG image domain, since the user is only able to change a few parameters at a time by clicking and dragging a shape on the canvas. The only advantage ad hoc brings is the possibility of structural updates.

In Script-n-Scribe, either mode should be viable. Assuming the user can only interact with one note at a time in the UI, at most only 3 values can be changed at a time. This should be easily handled by live mode. In contrast, ad hoc mode would likely see dozens of changes at a time. If possible, collapsing related changes would make synthesis much easier. That is, if the user modifies a note `(D 4 1/4)` to `(E 5 1/4)` then `(C 5 1/4)`, this should be registered as only 2 changes (D to C and 4 to 5) instead of 3 (D to E, E to C and 4 to 5).

4.2 Structural Updates and User Input Traces

The only significant different is the role of structural updates. In `sketch-n-sketch`, structural updates are reserved for only ad hoc mode, and even then are restricted to a single hard-coded instance. I found this to be unsatisfactory. Music composition involves creating many notes, rests and snippets. Forcing the user to add each and every one to the source code, instead of being able to write the new notes on the score, would defeat the purpose of prodirect manipulation and relegate the direct manipulation interface to merely tweaking values instead of using it as a tool equal to the programming interface.

Instead, I propose tracing user actions as a method for discovering structural updates. When the user triggers an object creation event — inserting a new note or rest, or creating a new (named) snippet — it is recorded as a *UI trace*. These are separate from the source code location and expression traces, and are used in a separate synthesis phase. After the user's other updates have been synthesized and substituted into the program, the UI traces are injected into the source code. There are two kinds of UI traces: value traces and snippet traces.

Value traces are created when the user creates a new note or rest. The trace records the information needed to insert the note into the source code. But what information is needed? We know that the new note was added to a snippet, because that is the granularity that the user works with during direct manipulation. One simple solution is to copy the trace from the "previous" note and use it for the new trace. Then the synthesis system inserts the new note after the note with the same location in the source code. Given that a snippet is an abstraction over the sequence combinator, the snippet $\{n_1, n_2 \ldots\}$ is evaluated as $n_1 :+: (n_2 :+: \ldots)$. The new trace n_u, which has the same trace as n_2 in this example, can be inserted by $n_1 :+: (n_2 :+: (n_u :+: \ldots))$. If the user inserts multiple notes, the location is carried through, and they are all inserted after the same note. Recall that the only requirement for a location trace is that it is unique, that no other numeric literal has that trace. Because UI trace integration takes place after synthesis occurs, this requirement can be relaxed. Additionally, the program is reinterpreted after each round of synthesis, so location traces are already transient. Stacking multiple values with the same trace will not matter once the program is reinterpreted.

Snippet traces are created when the user creates a new snippet. Each snippet is named at the time of creation, and this name is stored in the trace. That gives the synthesis system enough information to create the source representation of the new snippet (`name = {snippet...}`), but where should the snippet be inserted? As mentioned in section 3.1, the order of statements does matter, and the snippet must be inserted before it is used. We can assume that, since the user created the snippet via direct manipulation, they intend to include the snippet in `main`. Therefore, the lowest possible position for the new snippet is right before `main`. In general, the line before `main` should be a safe location to insert each new snippet.

In `sketch-n-sketch`, structural updates like these are only valid in ad hoc mode, when the user must explicitly synchronize the code. For UI traces, this is not necessary. In ad hoc mode, the user's cumulative changes are recorded. For example, if they create a note and then modify it, this is recorded as a single creation event, but the final note is recorded instead of the initially created note. In live mode, the note will have been added to the program before the user modifies it, since in live mode synthesis is triggered on every change. But for efficiency purposes, ad hoc mode may be a better default for Script-n-Scribe.

Unfortunately, this still is not sufficient. If when a note is created it takes the location of the note before it, what happens when a note is added to a newly-created empty snippet? Empty snippets do not have a location. (Actually, empty snippets are undefined. This language issue is discussed in section 6.1) One solution is to give them a default position, such as (-1, 0) for the first empty snippet, (-1, 1) for the second, and so one. As notes and rests are added to the new snippets, they borrow this default location, which the synthesis algorithm uses to insert them into the new snippet during synchronization. The default positions will be eliminated when the program is printed after synthesis, since the printer does not care about the traces at all. When synthesis is triggered next and the program is parsed again, the new notes and rests will all have appropriate locations.

5. Implementation

This section gives a brief overview of the implementation details of Script-n-Scribe. More details can of course be found in the source code.

5.1 Breve

Since I was already familiar with it from prior projects, I implemented Breve using the Parsec library [7]. As mentioned previously, the musical aspects of Breve, including the pitch classes, sequential and parallel combinators, and performance are handled by the Euterpea library [11].

The separation of integers and floating-point numbers is deliberate. It is partly due, I admit, to my own preference for stronger types. However, the main deciding factor was the Euterpea back-end. Euterpea's notes are constructed using (Pitch Class, Octave) pairs, where

"Octave" is a synonym for "Integer." A non-integer octave does not make any sense. If floating-point octaves were allowed in the program, then they would have to be converted to integers at some point anyway. This would be invisible to the user — an automatic cast.

5.2 Trace-based Synthesis

When synthesizing, each trace has one or more locations, and the synthesizer attempts to synthesize a new value for each possible location. Assuming the synthesis doesn't fail, each trace therefore produces one or more possible substitutions, which must be combined with substitutions for all the other traces to create the full update that will be applied to the original program. `sketch-n-sketch` decides which substitutions to use based on the mode: ad hoc mode asks the user, while live mode selects substitutions so that as many unique locations are used as possible, a sort of round-robin approach. The paper suggests that this decision is made before synthesis is performed.

In Haskell, lists have two meanings: the first is the common indexed collection of elements, and the second is a collection of the results of a nondeterministic computation. To put it another way, the synthesis result for each trace is one of several equally possible values. When Script-n-Scribe synthesises results, it leverages this model of the list to produce all possible substitutions before combining them into a final substitution. Then, based on whether it was called in ad hoc or live mode, it will process the list further before returning it to the caller.

5.3 User Interface

The user interface is implemented in Threepenny-GUI, a functional reactive programming library that creates GUIs with HTML and Javascript, serving them from a local server. I chose this library for a couple reasons: Javascript is still better than using any other GUI library, and it's much easier to make a UI that looks good thanks to CSS.

Unfortunately, due to a lack of experience and a lack of time, I was unable to fully realize the UI for Script-n-Scribe. The current version is able to perform basic synthesis, but only by simulating direct manipulation.

6. Known Issues

Given that this was my first time implementing a large, home-grown DSL, there a number of issues that need to be addressed. This section serves as an acknowledgment of those issues and a small discussion of how they may be resolved in the future.

6.1 The Question of the Empty Snippet

What is an empty snippet? Like an empty list, it is written as `{}`, which parses correctly but causes the evaluator to throw an error. A snippet with only one element is fine, and when evaluated returns just that element. The snippet is evaluated using the sequential combinator, which must take two values. Euterpea uses a rest with duration 0 (the empty rest, if you will) to represent an empty musical value. Interpreting the empty snippet as `(rest 0)` is a valid option for Breve, too, especially considering how empty snippets are used in user interactions. Instead of assigning a location to the snippet, we give it to the empty rest, and additional notes just lift it from the empty rest. Care should be taken that the empty rest is removed before the program is printed for the user to see, as it is essentially just visual noise.

The empty rest solves the question of empty snippets for user interaction. But what happens if the source program contains an empty snippet? We could still use the empty rest, but the rest still has a duration, which is a numeric literal, which means there needs to be a trace. The parser only records traces for numeric literals, not all expressions. Several solutions exist here, such as having the parser be aware of empty snippets and rewrite them as snippets containing an empty rest; or perhaps adding a dummy location and trying to remove the empty rest at every possible location. (e.g. if the empty snippet has an element added to it, drop the empty rest.)

6.2 Handling Errors

Errors can occur at dozens of points when parsing and evaluating a program. Any malformed input will cause the parser to fail. Evaluation may fail due to type errors, undefined variables, or any poorly written code. Robust error handling is absolutely required for a language to be used successfully by more than just its creator.

Unfortunately, Breve does not have that kind of error handling. In the beginning, it was sufficient to call `error` when something happened and let Haskell take care of it. This worked until the UI was added, at which point errors manifest as the UI becoming suddenly unresponsive with no indication of failure. Parsing, which uses the parsec library, provides error messages whenever the parser fails, but ensuring that useful error messages are presented can be difficult.

Haskell does have better standard error handling than what I have used so far, and I just need to take advantage of it. Adding more error checking will also help. For example, Breve does not check if a function has multiple parameters with the same name, which should be an error.

7. Future Work

Ultimately, my goal for Script-n-Scribe is a full-powered music composition system: multiple staves, multiple voices and instruments, annotations, parameters like tempo and dynamics, and so on. This is obviously a ton of work. The following sections describe useful, short-term and in some cases necessary work that will lay the foundation for the future of Script-n-Scribe.

7.1 Breve

While Breve is usable as it is, there are several features that would be very helpful. At the top of the list is a type system. As it stands now, Breve is not necessarily a strongly-typed language, but it at least has strong opinions about types. It does not care if your lists are heterogeneous, though it should, but it certainly won't let you fill a note with three pitches instead of a pitch and two numbers. An explicit, static type system would help users create correct programs, and hopefully improve evaluation.

Another useful feature would be more complex math. Currently, Breve can perform basic mathematics: addition, subtraction, multiplication and division. Implementing more complex functions, like trigonometry, would be supremely difficult to implement natively. My proposed solution is to implement the additional math functions as built-in functions, essentially a cross between operators and function calls. Syntactically, they look like function applications (e.g. `y = cos(x)`), but when evaluated they are passed to the equivalent Haskell functions.

7.1.1 Evaluation

Evaluation is quite complete at this point, besides the two issues listed in section 6. Another useful feature would be eliminating the statement order requirement. As it currently stands, all of the variables used in an expression must already have been evaluated before that expression is evaluated, or else evaluation will fail. One solution is to build a list of names in the program before evaluating, and have the evaluator recursively process the list to evaluate all names as they are needed. This could lead to an infinite loop where two expressions refer to each other without either being resolved, but it would be possible to detect this situation by marking which names have been visited but not evaluated.

7.2 Trace-based Synthesis

The most significant work left for trace-based synthesis is adding the user trace system described in section 4.2.

7.3 User Interface

Obviously, the GUI needs to be implemented. This is in my opinion the next big step for the project, before any other issue or future work is attempted. The file `doc/ui_plan.md` contains a discussion of the full details of the UI, including possible third-party libraries.

References

[1] Chris Channam and D. Michael McIntyre. *Rosegarden: Music software for Linux.* 2015. URL: `http://www.rosegardenmusic.com/`.

[2] Ravi Chugh, Jacob Albers, and Mitchell Spradlin. "Program Synthesis for Direct Manipulation Interfaces". In: *ArxiV* (2015).

[3] Csound Development Team. *Csound.* 2015. URL: `http://www.csounds.com/`.

[4] Cycling '74. *Max/MSP.* 2015. URL: `https://cycling74.com/products/max/`.

[5] Jeffrey Hass. "Introduction to Computer Music: Volume 1". In: Indiana University, 2013. Chap. 5. URL: `http://www.indiana.edu/~emusic/etext/digital_audio/chapter5_digital2.shtml`.

[6] Paul Hudak. *The Haskell School of Music – From Signals to Symphonies.* (Version 2.6), Jan. 2015.

[7] Daan Leijen, Paolo Martini, and Antoine Latter. *Parsec: Monadic parser combinators.* 2015. URL: `https://hackage.haskell.org/package/parsec`.

[8] MakeMusic. *Finale Music Notation Software.* 2015. URL: `http://www.finalemusic.com/`.

[9] John Roeder. "Pitch Class". In: *Grove Music Online, Oxford Music Online.* Oxford University Press, 2015.

[10] Avid Technology. *Sibelius.* 2015. URL: `http://www.sibelius.com/home/index_flash.html`.

[11] The Yale Haskell Group. *Euterpea.* 2015. URL: `http://haskell.cs.yale.edu/euterpea/`.

[12] Ge Wang. *ChucK.* 2015. URL: `http://chuck.cs.princeton.edu/`.

[13] Ge Wang and Perry Cook. "ChucK: A programming language for on-the-fly, real-time audio synthesis and multimedia". In: *Proceedings of the 12th Annual ACM international conference on Multimedia.* 2004.