# Introducing Circa: A Dataflow-Based Language for Live Coding

Andrew Fischer
Shutterfly, USA
Phoenix, AZ USA
andy.fischer@gmail.com

***Abstract*—In a live programming environment, the state of the running program is available during the editing process. An ideal live programming system should be able to harness the live program to offer improved abilities for code creation and manipulation. We introduce Circa, a language and platform designed to address this need. We argue in favor of a dataflow-based model of computation, and we show how this format enables useful methods of code inspection and manipulation. We present a framework based on the backpropogation algorithm that allows the user to manipulate their program by expressing a desire against the program's result. We discuss how these code editing abilities can combine to produce a highly effective environment.**

***Index Terms*—Live coding, dataflow programming.**

## I. Introduction

In a live programming environment, the user creates and modifies code while their program is running. One of the defining advantages of this setup is that the live program can be used as an aid during the code editing process. We can annotate the source code view with runtime information, such as the most recent result for a certain expression. We can also use the running program as a sort of magnifying glass, which shows us which sections of the code are relevant for the current state. This is a fundamental feature of Smalltalk-based environments [1][2], where the user can click on a graphical object, and navigate menus to see that object's definition. In an ideal live coding environment, code and runtime are intertwined, and the user can seamlessly jump between the two. Following these principles, we attempt to build a system from scratch that can best support a live editing workflow.

A language's design and implementation can often make this kind of runtime introspection difficult. In some cases, the association between runtime data and source code is not preserved during the compilation process. Even if the association is present, it may be difficult to communicate it to the user in a clear way. For example, if we would like to ask the system, "how was this particular value computed?" If the program uses a series of side-effecting steps that manipulate shared mutable state, then it can be difficult to say exactly which steps were responsible for that value. And, even if the language is pure, the use of too many higher-order abstractions might cause the answer to be practically inscrutable.

To that end, we need a programming model where code is highly introspectable and understandable. We choose a dataflow-based programming model, where a program is represented as a directed graph of terms. Each term has a list of inputs, and a function that specifies how to compute the output value. A function may be defined as a nested graph of more terms, or it may be a simple atomic operation. A function may also have an external effect, as long as its result value is purely computed from its inputs.

With a dataflow model, we innately have a greater ability to introspect on our program. For a given expression, we can always trace upwards to see where its inputs came from. We can show the user how a value was computed by showing the relevant function and input values. We can also freely reevaluate a section of code. A dataflow diagram also lends itself well to visualization, as demonstrated by the success of visual code editors such as PureData [3] and Max [4].

A dataflow-based code model has some drawbacks. A major problem is of expression: it's difficult to architect a large program as just a series of simple pure expressions. A quote by Alan Perlis is relevant here: "Purely applicative languages are poorly applicable." An area for future research is expanding on this language to support more expressiveness, without losing the properties that allow for deep runtime introspection.

In the remainder of this paper, we will present compelling methods of code editing that are made possible by a dataflow model.

## II. Flow-Based Introspection

We present a hypothetical scenario where the user is writing code to draw a simple scene (see figures 1 and 2). Our goal is to enable an interaction model where the user can click on the drawing in order to inspect and modify their code.

The process starts with a mouse click on the rendered scene. The first thing the runtime does is to determine which call to draw_sprite() is associated with that mouse position. We can determine this by reexecuting the code in a special pure-only mode, where all side-effecting functions are skipped. During this special evaluation, the runtime observes all of the calls to draw_sprite(), and it checks the position of each sprite against the mouse position.

```
1  groundY = 200
2  draw_ground(groundY)
3
4  bigTree = image('big_tree.png')
5  littleTree = image('little_tree.png')
6
7  bigTreeX = 50
8  littleTreeX = 150
9
10 bigTreePosition = [bigTreeX, groundY - bigTree.height]
11 littleTreePosition = [littleTreeX, groundY - littleTree.height]
12
13 draw_sprite(bigTree, bigTreePosition)
14 draw_sprite(littleTree, littleTreePosition)
```

*Figure 1. An example program*

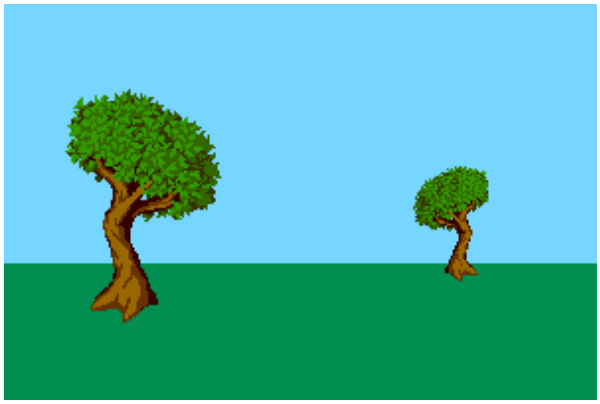

*Figure 2. The rendered result of our example program.*

```
1  groundY = 200
5  littleTree = image('little_tree.png')
8  littleTreeX = 150
11 littleTreePosition = [littleTreeX, groundY - littleTree.height]
14 draw_sprite(littleTree, littleTreePosition)
```

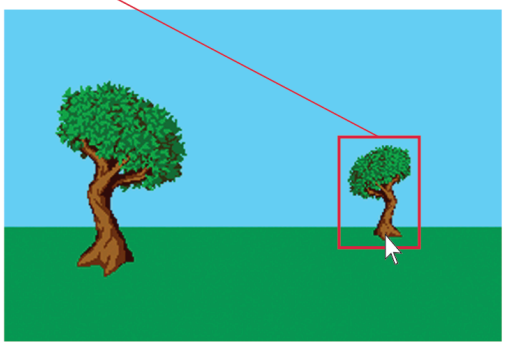

*Figure 3. Code view is filtered around one call to draw_sprite()*

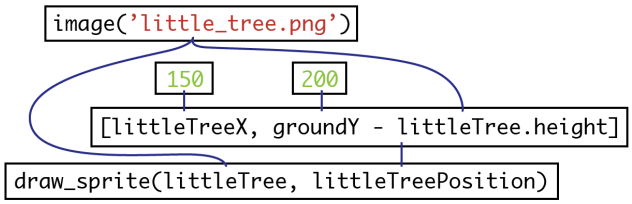


*Figure 4. A visual graph can be rendered from the above code.*

The runtime now has a copy of the program's intermediate state at the time when the relevant draw_sprite() call was made. We refer to this intermediate state as the "stack". It contains a list of stack frames, each containing intermediate values and links to the relevant code. The language's implementation allows stacks to be manipulated as first class values, including support for efficient duplication.

Using the stack, and taking advantage of our flow-based code model, we can present the user with a filtered view of the code. This filtered view only displays terms that were directly involved in the input values to draw_sprite(). From this view, the user has an easier time understanding the computation that went into this sprite's position. (see figure 3).

## III. Hybrid Textual/Visual Editing

Our code examples have thus far been displayed in a textual format, but having a highly-introspectable format allows us to present the same code as a graphical diagram (see figure 4). We don't consider there to be a difference between "textual" programming languages and "visual" ones, only a difference between textual and visual presentations. The corollary is that some languages are strongly suited for a certain kind of presentation. For various subjective reasons, we choose to use text as the primary storage format for Circa code.

After looking at the filtered code view, the user will likely want to make a code change directly to this view. This is possible because our stack contains links to the compiled source code data. The implementation stores code in a format that allows for easy implementation. Additionally, we implemented a whitespace-preserving decompiler which is able to reproduce the source text for a given code block. The user can save a modified code block back to well-formatted source code text.

## IV. Feedback on Flow-Based Code

The highly-introspectable flow-based model allows us to perform some even more clever methods of code manipulation. In the "feedback" scheme, the user expresses a desire against the result of a computation. A desire might be, "I want this result to be 5", or "I just want this result to be slightly smaller". The solver also receives various constraints, such as a restriction that only certain terms may be affected. Additionally the solver may receive hints, including the program's stack at the time when the desire was created. Taking all this input, the solver attempts to produce a code modification that would satisfy the requirements. The solver may answer that there are multiple solutions, and further specification is needed. The solver may also fail to find any solution.

This solving algorithm is inspired by the backpropagation algorithm [5]. An initial desire is expressed against the result of a computation. Then, we perform these two steps for each term involved in the computation:

1) Accumulation. All the pending desires for a given term are summed together.
2) Propagation/Resolution. We find an appropriate handler function, using multiple dispatch against the function and

the desire type. The handler examines the incoming desire. It may *propagate*, by sending a desire signal to one or more of the term's inputs. It may also *resolve*, creating a proposed code modification that would satisfy the incoming desire.

This process is repeated until all pending desires are resolved, or the algorithm fails.

Any step of the process has the potential to fail. The accumulation step may find it impossible to sum certain desires. The propagation/resolution step may not know how to respond to an incoming desire. Additionally, the entire process may produce a bad solution, even if each term was resolved successfully. Our solver is a general-purpose optimization algorithm, which is guaranteed to fail against some problems. We'll discuss the ramifications of this below.

Here are some examples of a "desire" signal. The system can be extended to support many desire types.

*RelativeValue*: The result should be a value that is X greater than before.

*NotBool*: The boolean result should be the opposite of what it was before.

*DiscourageEffect*: This desire is aimed at a side-effecting call, and indicates that the circumstances that lead to this effect should be avoided.

The accumulation step must combine incoming desires into one signal. If there are two or more, we must combine them in a way that's appropriate to the desire type. For example, if we have two *RelativeValue* desires, then combining them is to just perform an arithmetic sum of their values. Some combinations are impossible, such as *RelativeValue* with *DiscourageEffect*, and these would cause the solver to report failure.

Once the incoming desires are combined, a handler must use the summed desire. The handler is determined by dispatching against the desire type and the function. For some functions, the handler is straightforward. Consider the value() function, which is used in Circa to hold a literal value. The feedback handler for this function is relatively easy:

> If the desire is an *RelativeValue* or *NotBool*, produce a code modification that would change this value to the requested value.
>
> For other desires, report failure.

The value() function never propagates a desire, as it has no inputs. Next, consider the add() function, which adds two numbers. The feedback handler for add() is:

> If the desire is *RelativeValue*, check the constraints to see which inputs can receive feedback. If only one input can, then propagate an equal desire of *RelativeValue* to that input. If both inputs are able to receive feedback, then the situation is ambiguous. We can try to split the difference and send *RelativeValue* feedback to both inputs. If neither input can receive feedback, or there is a different desire type, report failure.

Some desire types are resolved independent of the actual function. For the *DiscourageEffect* desire, the strategy would be: locate the if-block that contains this term, and propagate a desire of *NotBool* against the conditional value for that if-block.

Finally, some functions may have a strategy that is independent of the desire type. Consider the cond() function which takes an boolean input, and returns one value if the boolean is true, and another if the value is false. This function's feedback handler would look at the value of the boolean at the time of the feedback signal. If this value is known, then the feedback handler can propagate the desire signal towards the input that was used at the time.

As mentioned in the add() example, there is often more than one way for the solver to satisfy a desire. Because of this, it's usually necessary to provide constraints to the solver. Most often, the solver needs to be constrained to only allow modifications to certain terms. The user can specify which terms to target using the code view interface. Alternatively, the source code itself might contain implicit or explicit hints, indicating that certain terms should or should not be targeted with feedback.

## V. Handling Feedback Failures

We have mentioned several ways in which our solver might fail, or not know what to do. This is not surprising, as our solver resembles a general-purpose equation solver, which is hard to do well and impossible to solve completely. With its numerous flaws, one might wonder how this system can be useful at all.

One observation is that the problems that the feedback solver will face tend to be relatively simple, mathematically speaking. Consider the code behind a typical real-world graphical user interface. When the program computes the position of each element, the actual math is most likely nothing more than a series of addition, multiplication and conditional operations. Our feedback-based solver has a good chance of success against this.

If the feedback solver does fail, we can try to recover. Our environment is innately live and has a high degree of user interaction. We can return to the user, present an annotated code view showing the solver's progress, and request more clarification or hints.

We might also be able to help the solver by statically annotating the code with hints. For example, our language could include a way to declare that B is monotonically increasing based on A. If the solver can't understand exactly how to convert a desire based on B to one based on A, it might be able to proceed using the hint. This issue could be considered a matter of coding style. Experienced users could develop a set of best practices for writing code that is amenable to feedback.

## VI. Feedback in Action

With this feedback-based solver in place, there are a few ways that we can exploit it. Returning to our above example, where our user is drawing a scene and they want to manipulate the code for a given sprite. As before, the process starts with the user clicking a sprite. The runtime would present the relevant code path. Next, the user manually selects which terms they want to manipulate. The feedback solver uses this selection as a constraint. Now, the user may click and drag the sprite. Internally, the runtime initiates a feedback operation for every mouse drag motion. It sends a *RelativeValue* desire (containing the distance of the mouse motion) against the position input to draw_sprite(). If successful, the sprite will move around the screen, exactly as the user would expect for a drag gesture. In this way, we have a form of programming by direct manipulation [6].

```
1   groundY = 200
5   littleTree = image('little_tree.png')
8   littleTreeX = 104.5
11  littleTreePosition = [littleTreeX, groundY - littleTree.height]
14  draw_sprite(littleTree, littleTreePosition)
```

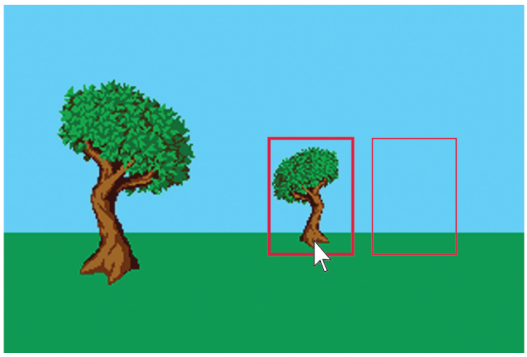

*Figure 5. Click-and-drag using feedback to modify a selected term.*

By introducing new syntax, we can further streamline the feedback-based editing process. Consider a situation where the user is writing new code, and they would like to tweak a certain value once their code is running. The user can indicate that a value is uncertain with syntax such as a "?" symbol. When the new code is submitted, the system will present the user with a feedback-based editing mode. The feedback operation is automatically constrained to only manipulate the expressions marked with "?". Coding can then be a hybrid process, where the initial structure of the code is written in as text, and then the "blanks" are filled in using the interactive environment.

## VII. Applications to Other Domains

The strategies that we have described are fairly generic, and would work in a variety of other domains, including non-live environments. A dataflow-based approach works especially well when the input and output values are easy for the user to understand and interact with. Our drawing example is a good fit for this approach, because directly manipulating the "output value" (the rendered scene) is an intuitive form of interaction. An example of a poor choice for this approach is a compression algorithm, where the output value (compressed data) is not easy to work with. However, even with the compression example, there is potential to decompose the algorithm's intermediate steps into a form that is amenable to introspection.

## VIII. Conclusion

Live programming has unique needs for language and runtime. By using a dataflow-based programming model, the system is able to more easily introspect on a running program. This introspection enables compelling new methods of code editing. We have shown how a runtime can show a trace view, displaying only the expressions relevant to one particular piece of data. We have presented a code manipulation strategy based on the backpropagation algorithm, where code is modified by expressing a desire against a computation result, and shown how it may be practically utilized.

Our plans for future work are to continue to refine these ideas for code manipulation, and continue exploring new ideas. One major focus for future work is expanding our language to support more powerful abstractions, without losing our code manipulation abilities. The backpropagation-based solver has demonstrated potential, but still has some unresolved issues before it can be used as a dependable tool. We will also continue improving the implementation, with the aim of releasing a complete environment for live programming.

## Acknowledgment

Thanks to Jennifer Seiler and Eric Daza for providing feedback and reviewing drafts of this paper. The tree artwork was created by TapSkill and all derived images are shared under the Creative Commons CC-BY-SA 3 license.

## References

[1] Goldberg, Adele; Robson, David (May 1983). Smalltalk-80: The Language and its Implementation.

[2] F. Olivero, M. Lanza, and M. Lungu. Gaucho: From Integrated Development Environments to Direct Manipulation Environments. FlexiTools Workshop, May 2010

[3] Puckette, M. S. (1997). Pure data. In: Proceedings of the International Computer Music Conference, pp. 224–227. International Computer Music Association.

[4] Danks, M. (1996). The graphics environment for max. In: Proceedings of the International Computer Music Conference, pp. 67–70. International Computer Music Association.

[5] Arthur Earl Bryson, Yu-Chi Ho (1969). *Applied optimal control: optimization, estimation, and control*. Blaisdell Publishing Company or Xerox College Publishing. pp. 481.

[6] Schneiderman, B. Direct Manipulation: A Step Beyond Programming Languages, IEEE Computer, Vol. 16, No. 8, pp. 57-69, 1983.