# Visualization of Dataflow Analyses

Catherine Chen Kevin Lim Myron Liu
University of California San Diego
hoc028@eng.ucsd.edu kyungyullim@ucsd.edu myronyliu@gmail.com

## Abstract

We demonstrate how dataflow analysis can be visually represented in the context of visual programming languages such as Google's Blockly, specifically for the analyses: *reaching definitions* and *constant propagation*. Our dataflow engine is such that the user can specify their own arbitrary analysis by defining new flow-functions. The goal is that such an interactive visual representation will be helpful for a young computer scientist learning about dataflow analysis for the first time, and encourage more direct experimentation with the idea of dataflow analysis.

***Keywords*** Visual Programming Languages, Dataflow Analysis, Education

## 1. Introduction

Graphical representation utilizes our complex visual system to make programs easier to understand and learn. Visual programming languages utilizes this ability and can improve students' ability to learn about complex programming concepts such as dataflow analysis and compiler optimization. Similar work has been done—as listed in the Related Works section—in other domains, and we will specifically be implementing and reproducing those ideas for compiler concepts. Further, interactivity allows a student to experiment and come up with scenarios and problems of their own, and verify the solution against our display.

With this goal in mind, we created a visual programming language dataflow visualization so that a young computer scientist being introduced to dataflow analysis, can experiment with the concept in a more direct manner then pen-and-paper. The technology framework we have chosen to test this idea is Blockly, a javascript visual programming framework.


*CSE 231 Spring 2015*, UC San Diego.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

## 2. Related Works

The effects and theory for visual programming languages have been explored by Green et al. and Myers [5][7]. There are many previous experiments that apply this insight such as Scratch, SNAP, or App inventors [1][2][3]. Evaluation of such systems have been done by others as well [4][6].

## 3. Implementation

### 3.1 dataflow representation

Our dataflow is represented internally as a dictionary with variables as keys and dataflow as values. For instance, in reaching definitions, our dataflow at the input (or output) of a given block in the program might be:

$$\{x : [1, 2, 4], y : [25], z : [5, 11]\}$$

signifying that the value for $x$ may come from blocks 1, 2, or 4. In contrast constant propagation is represented as follows:

$$\{x : 5, y : null, z : superConstant\}$$

As can be seen, the representation for the dictionary values may change depending on what is most suitable for the particular analysis. For user-defined analyses, the dataflow can take on any form so long as the corresponding flow-functions (which take in the dataflow as input) are consistent.

The set of analyses that our dataflow-engine accepts is also maintained in a dictionary: keyed on the *name* of each analysis. Suppose we have specified analyses *reaching definitions* and *constant propagation*. Then our dictionary of analyses would be:

$$\begin{aligned} reachingDefs : \{&flowFunc : function(block)\{...\},\\ &topFunc : function(workspace)\{...\},\\ &bottomFunc : function(workspace)\{...\}\}\\ constantProp : \{&flowFunc : function(block)\{...\},\\ &topFunc : function(workspace)\{...\},\\ &bottomFunc" : function(workspace)\{...\}\} \end{aligned}$$

Here, *flowFunc* takes in a block and computes the outgoing dataflow. *topFunc* and *bottomFunc* take in all the blocks in the code (hence the argument *workspace*) and compute

the lattice top/bottom respectively; these are defined so that in the case that *flowFunc* does not know how to handle a certain type of block, the outgoing dataflow for that block can always safely be set to *topFunc(workspace)*. Typically though, one shies away from reverting to the latticeTop. Reverting to a sub-lattice top usually suffices. For example, consider reaching definitions with blocks 1,2,3 and variables $x, y, z$. If $x$'s reaching definition becomes indeterminate at some point, it is far better to simply expand $x$'s dataflow to $x : [1, 2, 3]$ than setting the entire dataflow to $\{x : [1, 2, 3], y : [1, 2, 3], z : [1, 2, 3]\}$. This more refined behavior is handled by *flowFunc*.

In our existing implementation for reaching definitions and constant propagation, we always begin the analysis at the lattice bottom. Note that bottom in this case is represented by the empty dictionary $\{\}$; which is *not* the same as $\{x : [allBlocks], y : [allBlocks], z : [allBlocks], ...\}$ and $\{x : null, y : null, z : null, ...\}$: the respective tops for reaching definitions and constant propagation.

Our dataflow engine is separated into two parts. The first maintains a worklist of blocks to be processed, dynamically popping and adding elements as dictated by the worklist algorithm. The second computes outgoing dataflow given a block's incoming dataflow. This modular structure makes it easier to define custom analyses. All that is needed is an addition to the dictionary of analyses and corresponding specifications for *flowFunc, topFunc, bottomFunc*.

Once the dataflow for the entire program is generated, the results are displayed to the user. Each *outgoing* dataflow gets drawn off to the side of the block from whence it came. In addition, we label each statement block with its block ID, so that the correspondence between outgoing dataflow and block is completely unambiguous.

### 3.2 Worklist

The worklist algorithm bears some explanation. The iterative worklist algorithm traverses along a lattice as new information is found about the analysis. In order to create the initial worklist, the program's control flow graph is created, then traversed in depth-first post-order to push the blocks onto the worklist. Then, we iteratively run the flow functions until a fixed point for the whole program is reached.

Usually for a sequential block, a single run-through is enough and the analysis of it won't change. However, if inside a while loop we might possibly have to run the sub-blocks again. This check is done by whether or not a block's outgoing information has changed after running its flow function. If it has not changed, then we proceed down the worklist. However, if it has, then there is a chance that receiving block might have to be iteratively run again. Therefore in the case of a while loop, if it's output connection has changed, we merge it with the input connection of the while loop and put the first statement of the while loop back on the worklist, simulating an iteration until a fixed point is reached.

We are guaranteed termination here as we intentionally choose analyses that have finite lattices with monotonic flow functions and so we must reach the least fixed point.

### 3.3 Visualization

The visualization is handled mostly through blockly as a Scalable Vector Graphics XML description shown on an HTML page. So far, all information we add is a svg text element that either shows the block's ID, or the output dataflow at that point. The outgoing dataflow is indented to match the structure of the program to allow easier matching between the information and blocks.

## 4. Results

We ran our dataflow analysis on a number of test cases and verified them against manually calculated analysis results. The analysis supports control statements such as while and if/else blocks(Figure 1), as well as nested value evaluations(Figure 2). The results are displayed alongside the blocks and further clarified by displaying the ID of the block it belongs to.

One improvement that can be made is to draw a horizontal arrow between the outgoing dataflow summary to the bottom of the block with which it corresponds. This is actually something that we tried to implement, but for reasons that aren't particularly interesting, the result wasn't visually appealing. Usability improvements will be discussed further in the next section.

Additionally, our implementation for constant propagation only handles conditionals $if(x == expression)$ and $if(x != expression)$ where $x$ is a variable and $expression$ is some arbitrary expression that evaluates to a number or string (including expressions that involve variables themselves). This restriction limits the precision of the analysis. At the least, for constant propagation, we would like to include other boolean comparison operators such as $<, \leq, >, \geq$, which should be relatively straightforward to implement in the future. A more interesting example appears in figure 3 where the conditional $if(x \times 6 == 42)$ is an *implicit* expression. This sort of case is simple enough that it also begs to be handled. Nonetheless, the level of complexity we permit is largely subjective. All that is required is that unhandled conditionals are handled gracefully; in our case, we just default all referenced variables (in the conditional statement) to sub-dataflow *null*: the conservative option.

## 5. Future Work and Discussion

Although the results of the dataflow-analysis are displayed in an unambiguous manner, the fact that it is text-based leaves something to be desired. We wish to expand on this by improving usability through further visualizations, by permitting the ability to transform the program and see the analysis change interactively, and finally by testing the program and receiving user feedback.

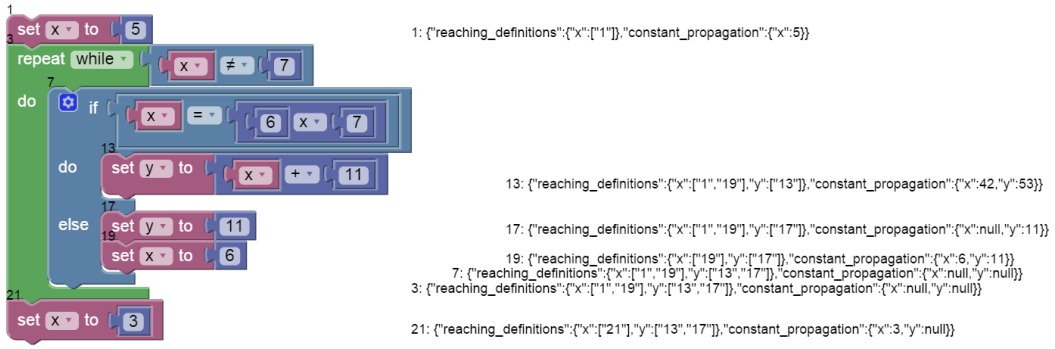


**Figure 1.** Dataflow analysis run and displayed on a generic program.

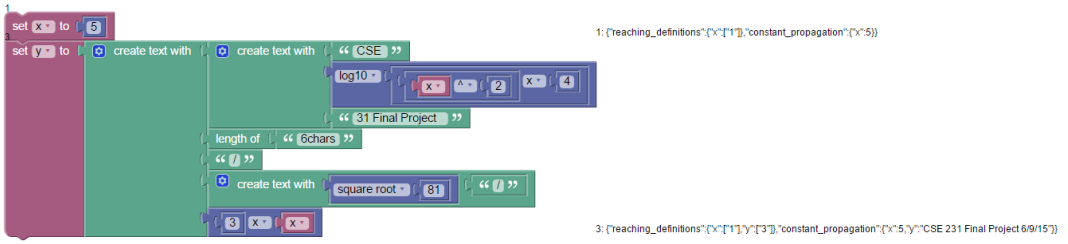


**Figure 2.** An example of constant propagation with heavily nested blocks.

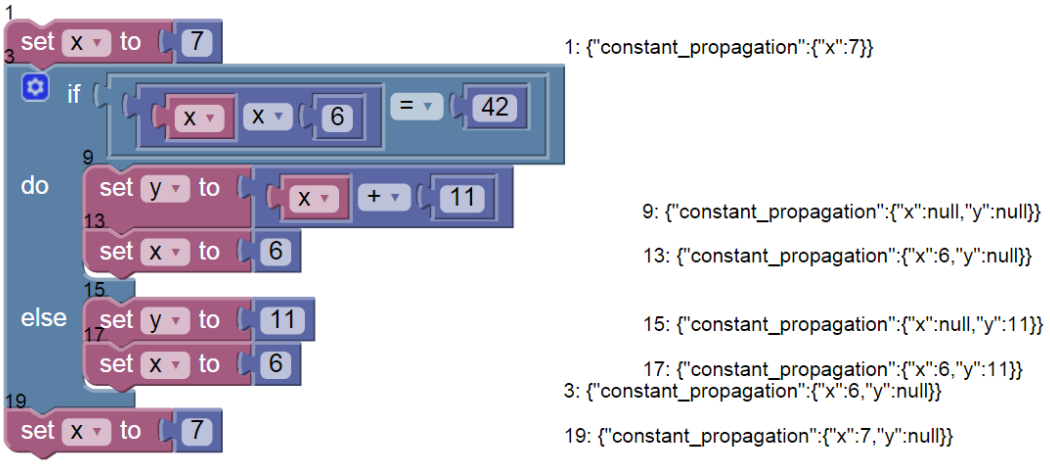


**Figure 3.** One limitation of our constant propagation analysis is that we do not currently handle implicit conditionals, such that we lose precision. As can be seen for the outgoing dataflows for block 9 and block 15, $x$ maps to $null$, which is excessively conservative.

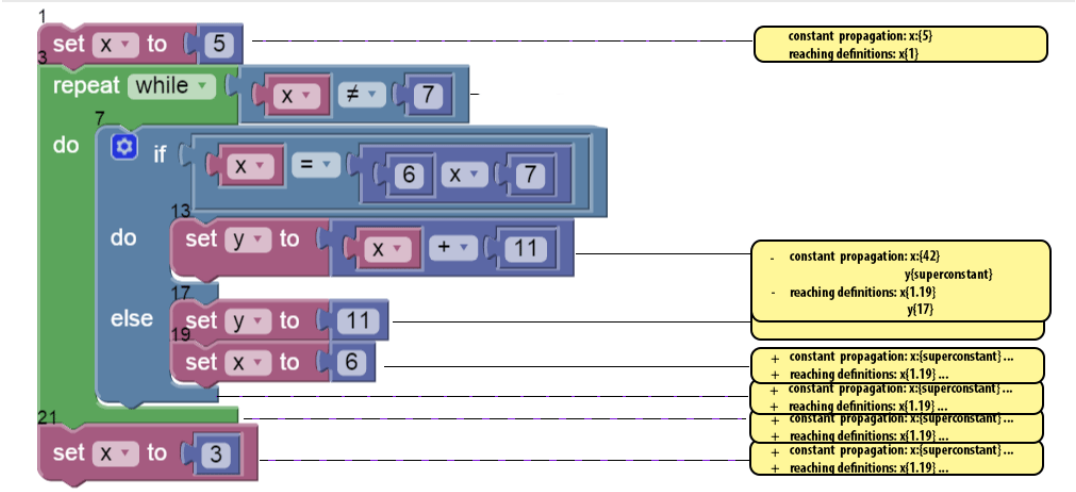


**Figure 4.** Mockup for expandable and collapsible information to reduce visual clutter.

### 5.1 Usability improvements

The current version is still far from our original mockups, and can be considered incomplete. Since a program can grow arbitrarily large, we cannot hope to display every dataflow information in a simple dictionary as we do now. Adding collapsible tabs per analysis is the first idea towards solving this issue(Figure 4).

The control flow graph of the program we construct for the analysis is implicit and only used internally. This representation can be expressed visually as in Figure 5, providing further visualization points to show our dataflow analysis and also helping the student learn the concept of control flow graphs.

To stay true to the spirit of visualizations, we wish to extend our information display from text-based to graphical. Further, the displayed information is static and cannot be interacted with whatsoever. Figure 6 shows an example of a block's dataflow information being visualized through edges rather than text interactively. Other interactive ideas include rerunning the analysis automatically based on events such as block creations or connection changes.

As aforementioned, one can specify their own dataflow analysis by introducing a corresponding dataflow format and flowfunction without worrying about issues like updating the worklist (which is automated). Currently, the only way to do this is by accessing the source. However, we hope to implement a *text field* in which the user can type their own data-formats and flow-functions. Blockly will then evaluate the contents, to run the custom analysis.

These mockups are still preliminary and would require further exploration.

### 5.2 Evaluation

Due to time constraints, we were unable to gather feedback and evaluate the performance of our application.

We plan to measure speed and engagement of our application [6] using the typical experiment framework in learning sciences by assessing students' performance in our learning environment against a control group that is given a more traditional instruction. Our hypothesis is that intermediate visual information makes students faster and more engaged about dataflow analysis and compiler optimization. A worksheet with identical problems will compare students who learn about dataflow analysis through our visual programming environment (the experiment group) and students who learn through a traditional lecture slide in comparable amount of time (the control group). The significance of our results can be measured with a student's t-test.

### 5.3 Interactive Optimization

With the dataflow analysis done, we are poised to execute a few optimizations across the program. Ideally this would be an interactive experience for the user as well and they can step through the optimization, observing which dataflow information allows which optimizations.

## 6. Conclusion

We presented a dataflow analysis engine for visual programming language that also displays its analysis results visually. Our goal of the project was to create an educational experience that made it clearer what dataflow analysis is and how it's performed. We hope the automatic dataflow analysis is helpful to users who try our visualization. There is still more work to be done, as currently the dataflow analysis simply gives the answer, and doesn't show how the in-

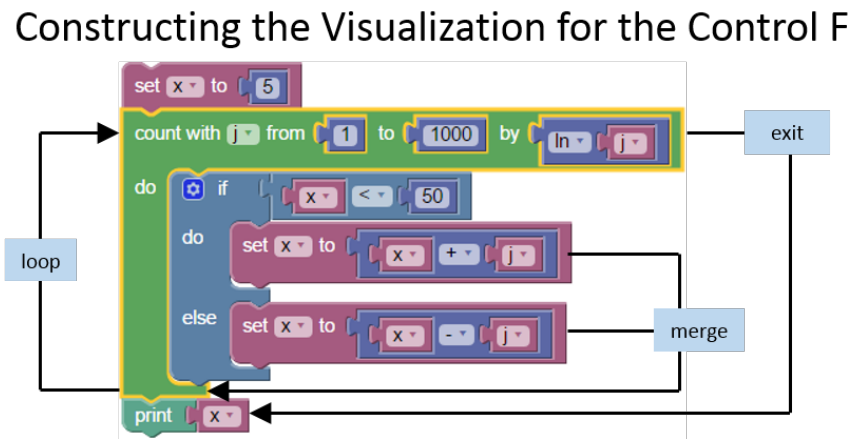


**Figure 5.** Control flow graph displaying additional blocks where control flow changes happen.

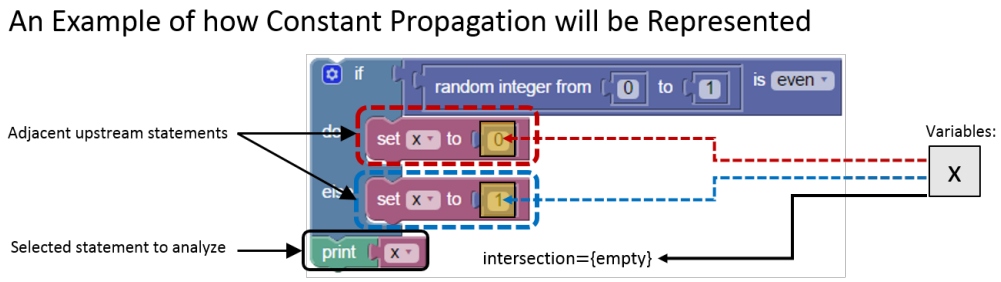


**Figure 6.** A selected node has its constant propagation information visualized by edges pointing to source blocks.

formation was constructed, which is vital to the educational experience. However, being able to modify the program and interactively see how such modifications change the analysis can be a powerful learning tool for a student to interact with.

Throughout this quarter we learned how to perform dataflow analysis at a high level. It is easy enough to step through code on paper, relying on human intuition to fill in the details. However, at a low-level, an abstract notion of dataflow analysis no longer suffices. This project gave us an opportunity apply our *fuzzy* concepts in a concrete setting. Having just barely scratched the surface, we now have a newfound appreciation for the discipline that is programming languages.

## Acknowledgments

We would like to thank Dr. Sorin Lerner and Alan Leung for their guidance and the CSE department for supporting this project.

## References

[1] App inventor. `http://appinventor.mit.edu/`.

[2] Scratch. `http://scratch.mit.edu/`.

[3] Snap. `https://snap.berkeley.edu/`.

[4] Sarah Esper, Stephen R Foster, and William G Griswold. Codespells: embodying the metaphor of wizardry for programming. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 249–254. ACM, 2013.

[5] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[6] Sorin Lerner, Stephen R Foster, and William G Griswold. Polymorphic blocks: Formalism-inspired ui for structured connectors. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3063–3072. ACM, 2015.

[7] Brad A Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990.