Language-Independent Interactive Data Visualization

Alistair E. R. Campbell, Geoffrey L. Catto, and Eric E. Hansen
Hamilton College
198 College Hill Road
Clinton, NY 13323
acampbel@hamilton.edu

Abstract

We introduce the *Language-Independent Visualization Environment (LIVE)* as a system for the visualization and manipulation of data structures and the computer programs that create and operate on them. LIVE interprets arbitrary programs containing arbitrary data structure definitions, showing diagrammatically the data that the process generates. It is language-independent in that a single program can be visualized in the syntax of multiple languages. LIVE is interactive in that not only does it show the effects of statements immediately as they occur at runtime, but it also generates new program source code automatically when the user manipulates the data on the screen. We anticipate that this tool will be most useful in a pedagogical setting such as a CS2 or data structures course, particularly with the introduction of pointers and linked structures.

Categories & Subject Descriptors

K.3 [Computers & Education]: Computer & Information Science Education - *Computer Science Education*

General Terms

Algorithms, Experimentation, Languages

Keywords

Visualization, Recursion, Scope, Data-Structures, CS2

1 Introduction

One of the most important issues faced by programmers in designing algorithms and data structures is their ability (or inability) to visualize the effects that statements and subprograms have on their data, and thus keep an accurate mental picture of complicated data structures. For many well-seasoned practitioners working with relatively simple data structures, visualization is done by simply imagining a picture of the data and reading the program source code. Intermediate programmers frequently visualize processes by drawing a picture on paper and tediously tracing each statement in a program. If they don't get it right the first time, they will probably find their errors eventually through this kind of off-line visualization. Beginners have the most difficult time with visualization because not only do they have the least experience with particular algorithms, they also have the added burden of not completely understanding the fundamental effects of the individual statements. They are often frustrated because they don't even know the basics of what they're supposed to be visualizing.

Another issue is the problem of multiple languages and paradigms of programming. A C++ programmer switching to Java (or vice versa) often has difficulty with what appears to be familiar syntax, but entirely different semantics. Well-educated computer scientists will quickly learn which principles and constructs are meant by the new syntax and adjust their programming style. Beginners, however, struggle at great length because, to them, the syntax is often the hardest part.

We seek to develop pedagogical tools that address these issues directly. Our premises are that the visualization of a program's underlying data is crucial to a student's understanding of the behavior of algorithms, and that seeing the same algorithm in the syntax of multiple languages improves understanding of statement and operator semantics from both familiar and unfamiliar languages. Student understanding will be enhanced by environments that allow them to manipulate data structures on the screen and to see immediately the effects of program statements as they are executed. This paper introduces one such environment, enabling the visualization of arbitrary user-defined data structures in arbitrary programs, and supporting syntax from multiple programming languages simultaneously.

2 Related Work

Previous work has demonstrated the benefits of visualization as an aid to student learning. Tango, Polka, and Samba are systems for building algorithm animations by way of augmenting existing programs with extra statements that drive the animations [5, 6, 7]. With Samba in particular, students are encouraged to be interactive in their learning by creating the animations themselves. The JAWAA and JFLAP systems provide for web-based data structure, algorithm, and

SIGCSE'03, February 19-23, 2003, Reno, Nevada, USA.

automata theory animations [4, 3]. The JVALL software animates the operation of linked lists in the Java programming language [2]. Other work in data structure visualization gets closer to the implementation details of a running process by interfacing with the Visual C++ debugger to display a program's variables diagrammatically, indicating pointer assignments, heap and stack allocations, memory leaks, etc. [1]

3 The LIVE system

We have developed LIVE (*Language Independent Visualization Environment*), a system that supports the visualization of arbitrary data structure definitions, multiple language syntax, recursive subprograms, and lexical variable scope. The user can invoke LIVE, enter a program in one language, and step through the statements of the program. While the program is running, the user can add statements to it, and can view it in the syntax of other languages.

3.1 Interaction

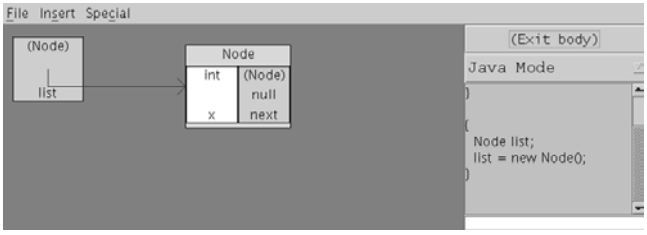
When the user starts LIVEGUI, the graphical interface to LIVE, he or she is presented with a blank frame (the *canvas*), onto which data can be placed. LIVEGUI allows the user to perform the basic operations of repositioning and resizing data on the canvas, and to select system preferences like font size, background color, etc.

In an accompanying panel on the right side of the canvas, a computer program can be displayed and edited. When the user selects **Run** from the file menu, LIVE parses the program, clears the canvas, and begins an interpretation of the syntax tree. Each program statement is displayed just before it is executed. The speed of this evaluation can be varied, or even set to zero, in which case the user can step through the interpretation manually. A *statement field* below the main code window allows a user to add new statements to be executed immediately and added to the program.

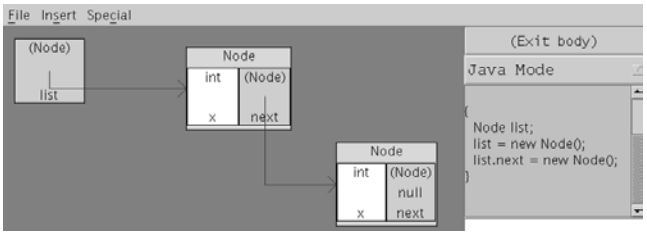
From the menus provided, the user can create new data types, instantiate named variables, assign pointer values, and allocate or delete data dynamically. In response to each interaction, LIVE generates a new source code statement that has the same semantics as the user's manipulation. As an example, consider the task of creating a circularly linked list with two nodes, in the Java syntax. An initial program is created with the node type definition and the declaration of `list`, a single object reference variable (pointer). The user runs the program in LIVE, placing that variable on the canvas:

To instantiate the first node, the user selects `list` with the mouse, and chooses Create New Dynamic Object... from the menu that appears:

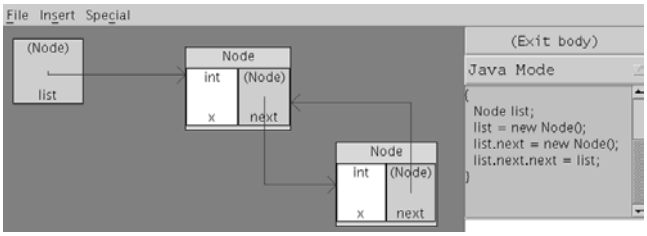
In the code window, the statement `list = new Node()` is added to the program and a new `Node` appears on the canvas.



Similarly, the user selects the `next` field of that node and allocates the second node. The statement `list.next = new Node()` is automatically generated:



The cycle is completed by two gestures: the selection of Assign pointer to... on the `next` field of the second node, and one click on the first node. LIVE adds `list.next.next = list`:



Thus, except for the the initial declaration of the named pointer variable, every statement in the program body is automatically generated by user interaction with data using the mouse.

3.2 Multiple languages

Multiple language modes support the display of programs and data types in LIVE. A computer program is represented in the LIVE system as a syntax tree. The same syntax tree notation is used in all language modes. The tree is created by a parser appropriate to the current mode. The parsers, one for each language supported, were produced with JAVACC, the Java Compiler Compiler [8]. Associated with the syntax trees are code generation algorithms that render the trees in any of the supported languages. Currently, LIVE supports C++, Java, and our own in-house language *Überlanguage*, reminiscent of Pascal. The latter could be considered LIVE's native language not only because its parser was developed

before the other two, but also because elements of its syntax are used internally by the LIVE system.

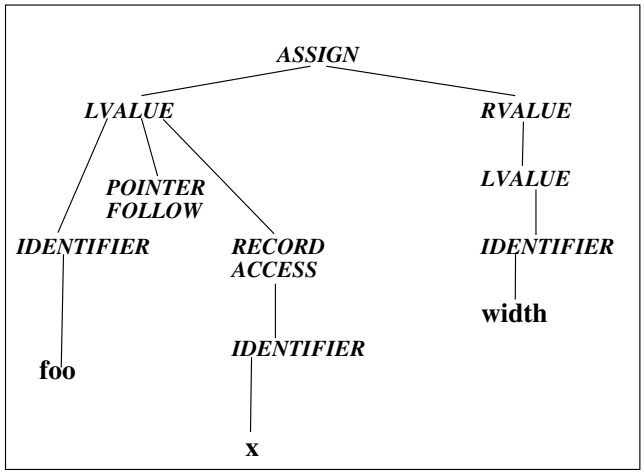
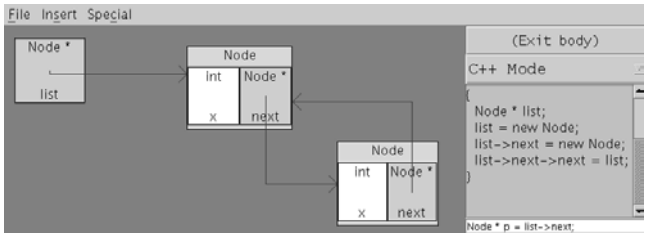


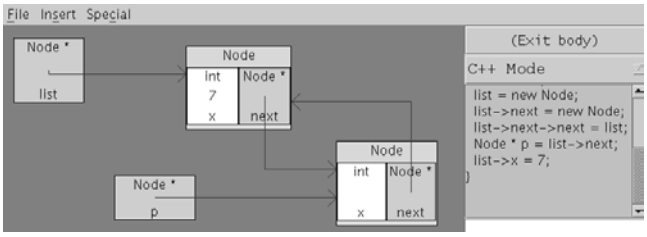
Figure 1: The LIVE syntax tree for the C++ statement `foo->x = width`, the JAVA statement `foo.x = width`, and the Überlanguage statement `fooˆ.x := width`

Generating code from the same syntax tree in multiple languages improves a student's understanding of syntax and semantics for assignments, particularly with respect to how pointers and structure accesses work. One of the illuminating effects of having syntax trees as the underlying code representation is that when the code for a subtree is generated in the target language, the simplest syntax is used. For example, if the user enters the C++ statement `(*foo).x = width`, it is parsed by the C++ parser into the syntax tree shown in figure 1. When that tree is generated in C++, the user sees `foo->x = width`. Likewise, if the user switches to Java mode, LIVE generates `foo.x = width`.

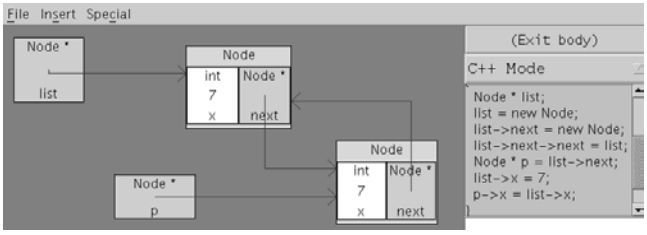
Returning to LIVE, still running our circular list example, the user switches to C++ mode using the selector above the code window. To declare and initialize another pointer `p`, the user enters C++ source code directly into the statement field in the lower right corner:



The user selects the `x` field in the first node, chooses Assign integer literal from the menu that appears, and enters the literal number 7 in a dialog box. LIVE generates `list->x = 7` in response:



Finally, the user selects the `x` field in the second node, chooses Assign integer from..., and then selects the `x` field in the first node, yielding `p->x = list->x`:



Note that the code generation algorithm has chosen the shortest paths to the data involved in the last two statements.

3.3 LIVE programs

LIVE supports data types for primitives, pointers, and records. The primitives are *boolean, integer, character, string,* and *float.* Arbitrary record data types can be defined by the user, as long as their fields are primitive, previously defined records, or pointers. A pointer is a reference to a datum of any other type, including another pointer type. Program data may be created at run time either statically, as variables in a program body, or dynamically, as the result of a dynamic allocation operation. LIVE maintains its own stack for keeping track of static variables, as well as a heap containing the dynamically allocated data.

The LIVE system runs its programs by interpreting them, leaving much of the semantic consistency checking until run time. Two examples of this are type checking for assignments and variable or record field lookup. If types don't match, or if a variable isn't in scope, LIVE signals an error condition and LIVEGUI notifies the user. Nevertheless, even after errors occur, it is often the case that the user can continue running the current LIVE program.

LIVE provides for the declaration and calling of subprograms, with call-by-value parameter passing. A subprogram may return a value to its caller. The ordinary static scope rules for variables from Java and C++ also apply in LIVE: a variable may be declared in any body and has scope until the end of the body. When data are instantiated, they become visible on the canvas until they are longer in scope. Subprograms may be recursive, often resulting in multiple instances of the same variable name on the screen at the same time. Two variables with the same name may have nested scopes, resulting in holes in the scopes of the outer variable. Figure 2 illustrates examples of scope holes and recursion in LIVE.

Other operations currently supported by LIVE include assignment, standard arithmetic, an operator returning a pointer to a datum, its inverse pointer following operation, a record field access operator, dynamic allocation

and deallocation operators, and the boolean-returning numeric comparison operators written in Java or C++ as {<,<=,==,>=,>}. The equality comparison operator (==) is overloaded for pointer comparisons as well. Control structures include selection (if-else) and iteration (while, for). Statements for interrupting control structures (return, break) are supported with approximately the same semantics as in Java or C++.

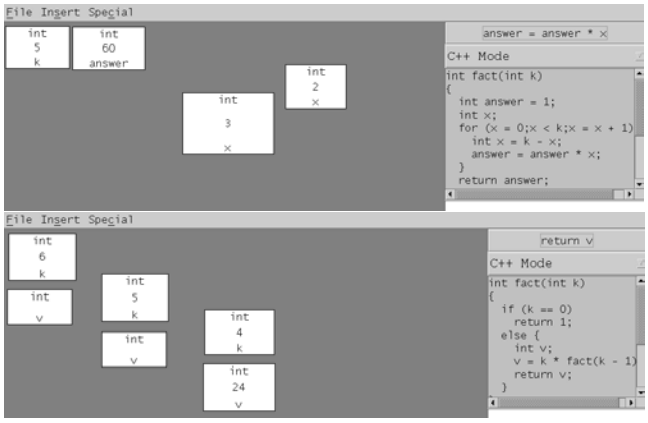


Figure 2: Computing factorial two ways.
In the top canvas: An example of nested scopes. The larger x in the center stays fixed; the other x moves around as it is allocated and deallocated repeatedly.
In the bottom canvas: An example of recursion. LIVE is starting to return back up the call chain of a recursive factorial function, with several instances of parameters and local variables waiting.

4 Discussion

LIVE currently supports operations in the imperative paradigm of programming, thus it is best suited to use with languages that employ sequence, selection, iteration as primary control structures, along with subprogram call as a means of abstraction.

In this paper, we are neutral on the question of whether an object-oriented approach should be taken in introductory courses. Our current system is procedural, rather than object-oriented, because the principles it is best suited to demonstrate—structured data, pointers, dynamic memory management, block-structured allocation and deallocation of variables, subprogram flow of control, and recursive subprogram calls—all have their roots in the procedural paradigm and are demonstrated most effectively without the added burden of issues like inheritance, method lookup tables, public vs. private data and methods, constructors and destructors, etc. We consider LIVE not as a means of visualizing the execution of a complete Java or C++ program, but rather as an aid to understanding how the basic operations within an ordinary function or object-oriented method change the data structures when they are executed.

Many of LIVE's features are valuable in pedagogical settings, both in and out of the classroom. In class, for example, a teacher can start LIVE with a program containing only the record declaration for a singly-linked list node, then alternate between manipulating the visualization and entering statements, asking individual students to predict the behavior. Once the class is comfortable with the basics, individuals or groups may be challenged to enter code into LIVE to generate a complicated data structure. Perhaps more importantly, after class, a student can use LIVE independently to reinforce the concepts initially shown in class.

One of the more distinctive features of LIVE is that it shows the user his or her data structures in the manner that the user desires, not in a manner computed by the system. This places the burden of data organization on the user, but the burden is reasonable on two counts. First, LIVE supports arbitrary programs and structures. Because many different data structures share similar type definitions, there is no way for the system to automatically infer a correct scheme for data placement. Consider, for example, that a binary tree node and a doubly-linked list node are structurally equivalent, each with a data field and two pointers, but a binary tree should probably not be displayed linearly. Second, LIVE is meant to be used as a learning tool. We want our students to be actively involved in observing a program run. When a datum pops up onto the canvas, we want them asking questions like *Why has it appeared now? What other data can I expect to see later? When will it disappear? Where should I put it in relation to other data?* Making the student be responsible for the neatness of the visualization helps them figure out what's going on more quickly than by simply watching.

5 Future Work

Initial student reaction to LIVE has been very positive. We have demonstrated the system to a number of computer science concentrators, all of whom agreed that its use would have increased their understanding of pointers, data structures, and recursion, had they been provided with LIVE at the time they were first learning these basic principles. LIVE has also been used successfully in the classroom to illustrate the different semantics of apparently identical Java and C++ variable declarations, and to show the effects of nested variable scope. LIVE will be used more extensively in the classroom in Fall 2002, to demonstrate pointers and recursion in a CS2 course. We plan to study in detail how the students and faculty make use of the system.

We are continually improving LIVE by adding new features to its core and by enriching the syntax supported by its language processors. Most notably, our immediate plans call for the implementation of reference parameters and array types.

LIVE will be expanded to include method declarations within structures and classes, to parse and respect the distinction between public and private object and class methods and data, and to support dynamic method lookup for object-oriented message passing.

Finally, difficult problems arise when some features in one language are not compatible with another language. For instance, C++ supports pointer arithmetic, but Java does not. Pure functional languages don't allow variable assignments. While we may never know how to unify all conceivable language properties under one visualization system, partial resolution of this general problem remains open to further research.

6 Availability

LIVE is available free of charge under the GNU public license. Its Java 1.2 source code and supporting documents may be found on the LIVE project website: `http://www.cs.hamilton.edu/~alistair/LIVE`.

7 Conclusion

We have introduced LIVE: the Language-Independent Visualization Environment as a tool for teaching and learning about arbitrary linked data structures, recursion, and variable scope. The main contributions are (1) that it has a language-independent framework for visualizing code, data and their types; (2) that it displays the effects of code execution diagrammatically while the program is running; and (3) that it automatically generates source code statements in response to user manipulation of the data visualized on the screen.

8 Acknowledgments

This work was supported, in part, by the Ralph E. Hansmann Science Students Support Fund at Hamilton College. Thanks also to Rick Decker and Julie Parent for their feedback on early versions of LIVE; and to Stuart Hirshfield and Peter Kimball for helpful comments on earlier drafts of this paper.

References

[1] Costigan, J., Wilhite, B., and North, C. Data structure visualization with visual debugger: A tool for automatic visualization of run-time data structures. Online. Internet. September, 2002. Available WWW: `http://infovis.cs.vt.edu/datastruct/`.

[2] Dershem, H. L., McFall, R. L., and Uti, N. Animation of Java linked lists. In *Proceedings of the* 33^{rd} *SIGCSE Technical Symposium on Computer Science Education* (2002), pp. 53–57.

[3] Gramond, E., and Rodger, S. H. Using JFLAP to interact with theorems in automata theory. In *Proceedings of the* 30^{th} *SIGCSE Technical Symposium on Computer Science Education* (1999), pp. 336–340.

[4] Pierson, W. C., and Rodger, S. H. Web-based animation of data structures using JAWAA. In *Proceedings of the* 29^{th} *SIGCSE Technical Symposium on Computer Science Education* (1998), pp. 267–271.

[5] Stasko, J. TANGO: A framework and system for algorithm animation. *IEEE Computer 23*, 9 (1990), 27–39.

[6] Stasko, J. T., and Kraemer, E. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing 18*, 2 (1993), 258–264.

[7] Stasko, J. T., and Lawrence, A. Using student-built algorithm animations as learning aids. In *Software Visualization: Programming as a Multimedia Experience*, J. T. Stasko, J. B. Domingue, M. H. Brown, and B. A. Price, Eds. The MIT Press, 1998, pp. 419–438.

[8] WebGain, Inc. Java compiler compiler (JAVACC)–the java parser generator. Online. Internet. [September 2002]. Available WWW: `http://www.webgain.com/products/java_cc/`.