Dynamic evaluation trees for novice C programmers

Matthew Heinsen Egan[1] **Chris McDonald**[2]

School of Computer Science and Software Engineering
University of Western Australia
Crawley, Western Australia 6009
Email: [1]m.heinsen.egan@graduate.uwa.edu.au
[2]chris.mcdonald@uwa.edu.au

Abstract

The *dynamic evaluation tree* is a method of visualizing expression evaluation that annotates a program's source code with expression results. It is intended to reduce students' visual attention problems by removing the need to alternate between disparate source code and expression evaluation windows. We generalise the dynamic evaluation tree to support arbitrary expressions in the C programming language, and present the first ever implementation for a novice-focused program visualization and debugging tool.

Keywords: Novice programmers, debuggers, software visualization

1 Introduction

Expression evaluation can be difficult for novice programmers to comprehend. An incomplete understanding of expression evaluation may make it exceedingly difficult for novices to identify and correct malformed expressions in their own code. In a multi-institutional study of novice debuggers, Fitzgerald et al. (2008) found that the most difficult bugs for their subjects to find and fix were arithmetic bugs (in particular) and malformed statement bugs (in general). Effective visualization of expression evaluation may assist novice programmers to construct knowledge of expression evaluation, including the behaviour of individual operators, and to debug programs containing malformed expressions.

Brusilovsky & Spring (2004) discussed a tutoring system designed to assist students learn expression evaluation in the C programming language, stating:

> "For the students in our programming and data structure courses based on C language, expression evaluation is one of the most difficult concepts to understand. They have problems with both understanding the order of operator execution in a C expression and understanding the semantics of operators."

The web-based system, WADEIn, visualizes the step-by-step evaluation of expressions consisting of mathematical and logical operators with `int` and `double` type variables. More than 80% of students felt that the system helped them to understand C operations.

Many existing software visualization systems use a dedicated "*expression evaluation*" area to visualize the individual operations performed during an expression's evaluation (e.g. Jeliot 3, as presented by Moreno et al. (2004); and The Teaching Machine described by Bruce-Lockhart et al. (2007)). Animation is commonly used to relate operations to the expression's source code, and operands to memory visualizations. For example, if an evaluated operator's operand is a variable, then the variable's value might "fly in" from the memory visualization.

Lahtinen & Ahoniemi (2009) introduced the "*dynamic evaluation tree*" for visualizing expression evaluation by annotating above or below a program's source code, e.g.:

```
int c = a + b;
        1   2
        └───┘
          3
```

This concept was primarily motivated by the results of an eye-tracking study of Jeliot 3 users, which found that novice programmers "*either switch their visual attention repeatedly between different windows or concentrate all the time on one of the windows*" (Lahtinen & Ahoniemi 2009). The dynamic evaluation tree is intended to integrate expression evaluation and source code representation, thus reducing the switching of visual attention required by novice programmers. Lahtinen and Ahoniemi discussed the potential of adding the dynamic evaluation tree to the VIP C++ program visualization system, but unfortunately this work has not been continued.

Annotations in a dynamic evaluation tree maintain a visual relationship to their associated source code, as opposed to animated visualizations which only briefly show this relationship (e.g. by having the relevant source code "fly in" to the evaluation area). This explicit visualization of the expression evaluation's history may reduce the need for students to step backwards and forwards, and clarify the relationships between individual operations.

This paper discusses our implementation of a dynamic evaluation tree for the novice-focused program visualization and debugging tool *SeeC*. Section 2 generalises the dynamic evaluation tree to support arbitrary expressions in the C programming language. Section 3 describes our implementation. Section 4 discusses integration with SeeC's other components. Section 5 compares our implementation with traditional visualizations of expression evaluation. Section 6 discusses limitations in our implementation and identifies future work. Finally, Section 7 summarizes our discussion.

2 General C programs

Despite its age, the C programming language still holds an important place in computing education. While few traditional Computer Science courses teach C as an introductory programming language in their foundation years, C remains important in the later teaching of operating systems and computer networking. C still enables students to understand the close relationship between programming languages and hardware in increasingly important subjects such as robotics, embedded systems, and wearable computing, and these subjects are often required by students other than future computer scientists. Novice C programmers are not necessarily novice programmers, and those whose entire exposure to programming has been through safe languages still have to address many challenges. Many of the traditional problems with C, such as its practice of leaving much as "defined to be undefined" and the challenges of writing portable code across disparate operating systems and architectures, have been addressed by detailed official standards, shifting the pressure to those teaching C to do so well.

The simplicity and familiarity of the dynamic evaluation tree is a great strength. It provides a concise, clear representation of complex expression evaluations. Implementing the dynamic evaluation tree for SeeC required us to support arbitrary expressions in the C programming language, introducing several complicating details. This section discusses these complications and the approaches that we employed to ensure that the dynamic evaluation tree retains its conciseness, clarity, and, we believe, usefulness.

The simplest problem is that an annotation's text may be wider than the annotated expression's source code. This may obscure the visual relationship between the annotation and source code, and could lead to overlapping annotations. We prevent this simply by truncating annotation text to the width of the expression's source code. Students can view the complete text by hovering the cursor over the annotation.

The dynamic evaluation tree is designed to annotate a single line of source code, but students are free to write an expression over multiple lines. This may be uncommon in novice programmers' code, but our general purpose implementation must account for it. Our straightforward solution is to reformat the expression's source code, displaying it on a single line while the dynamic evaluation tree is active.

The C programming language's *preprocessor* may also necessitate the use of modified source code to represent expressions, as a single macro may expand to multiple sub-expressions. If each expression had at most a single child, we could simply stack the annotations. For example, consider a typical implementation of the `NULL` macro:

```
#define NULL ((void*)0)
```

NULL
integer literal: 0
cast: 0x0

For more complex macros the visualization will become increasingly crowded. As an example, consider the `sys/stat.h` header's `S_ISREG` macro, defined by The Open Group Base Specifications Issue 7 thus[1]: *"The value `m` supplied to the macros is the value of `st_mode` from a `stat` structure. The macro shall evaluate to a non-zero value if the test is true; 0 if the test is false."* A typical implementation of this macro is:

```
#define S_ISREG(m) (((m) & S_IFMT) == S_IFREG)
```

[1] http://pubs.opengroup.org/onlinepubs/9699919799/

Visualizing the complete tree created by using the macro `S_ISREG` would expose students to unnecessary, potentially confusing implementation details. Thus it may be best to employ a black box representation by restricting the visualization to the "input" and "output" nodes: in this case, `m` and the result of the `==` operator, respectively. Conversely, it should be possible for students to observe the behaviour of code produced by their own macros: showing the preprocessed code will allow students to observe their macro's expansion, and a dynamic evaluation tree visualizing the resulting expression's behaviour.

In the C programming language an expression may designate an object; such expressions are termed *lvalues*[2]. For example, in line 4 of Listing 1 the expressions `total`, `iptr`, and `iptr[i]` are lvalues. An expression which does not designate an object, for example the expression `total + iptr[i]`, is commonly referred to as an *rvalue*[3].

Listing 1 Summing an array of `int` values

```
1 int sum_ints(const int *iptr, size_t n) {
2   int total = 0;
3   for (size_t i = 0; i < n; ++i)
4     total = total + iptr[i];
5   return total;
6 }
```

Some expressions require an lvalue, e.g. the unary `&` operator produces the address of the designated object, and the `++` operator increments the value stored in the designated object. For most other uses an lvalue is converted to the value stored in the designated object, e.g. `iptr[i]` in Listing 1. In terms of the language implementation we might consider this to represent the value being loaded from memory. The behaviour of such lvalues poses a question for the visualization of dynamic evaluation trees: should we show the designated object, the value that was stored in the designated object when the expression was evaluated, or both? An explicit relationship to the designated object will allow students to see where values are coming from. This may be particularly useful for array accesses and pointer dereferences. However, showing the value stored in the designated object may be confusing if the value changes after the expression is evaluated, for example:

```
number = 10 / number ;
             2
         5
```

When this assignment expression is completed the value `5` will be stored in the object designated by `number`. However, the value of the `number` expression on the right hand side should still be `2`, otherwise the division's result is nonsensical. Our approach is to show two nodes: one for the lvalue, and one for the rvalue it was converted to during evaluation. The lvalue is annotated with descriptive placeholder text rather than the designated object's value. When the student moves the cursor over this annotation, the designated object is highlighted in SeeC's standard memory visualization.

Expressions with `struct` or `union` types are difficult to represent within an annotation, as they may contain numerous fields and values, thus causing the

[2] ISO/IEC 9899:2011 (The C11 Standard) §6.3.2.1.1
[3] ISO/IEC 9899:2011 uses the term "value of an expression".

textual representation to be far larger than the expression's source code. If the expression is an lvalue then we again show a placeholder and direct students to a memory visualization for the complete value. This is not possible for rvalue expressions, so we truncate the annotation when necessary and show the complete value when the student hovers the mouse cursor over the node.

Pointers, often described as a threshold concept in Computer Science (Boustedt et al. 2007, Rountree & Rountree 2009), are a source of great difficulty for novice C programmers, and so it is essential to effectively visualize pointer type expressions. The raw value of a pointer is generally not important for novice C programmers, rather they are concerned with whether pointers are valid and which objects they reference. Displaying the value of the referenced object could visualize this information, but might cause dangerous misconceptions about the semantics of pointers. We handle this similarly to lvalues: the node's annotation contains placeholder text, and when students move the mouse cursor over the node the referenced object is highlighted in SeeC's memory visualization. The placeholder text indicates whether the pointer is valid, invalid, opaque, or `NULL`.

3 Implementation

We implemented a dynamic evaluation tree as an extension to the SeeC project: a system for novice C programmers that performs execution tracing with automatic runtime error detection, and provides program visualization of the recorded execution traces, as described by Heinsen Egan & McDonald (2014). SeeC itself is built upon the Clang project[4]: a modular collection of libraries which implement a front-end for compiling C, C++, Objective C, and Objective C++, but are also designed to support diverse uses by external clients. Students reviewing an execution trace can step forwards or backwards to any point in the process' execution. The SeeC system provides a "*recreated state*" of the process, which we use to generate the dynamic evaluation tree.

The "*recreated state*" of the function that was executing provides us with the "*currently active*" statement, which is either partially evaluated or has just completed evaluation (in which case it may have produced a value). If this statement is an expression then we walk up Clang's Abstract Syntax Tree to find the "*top-level*" expression, i.e. the first node whose parent is not also an expression. The top-level expression is the root of our dynamic evaluation tree, ensuring visualizations remain consistent during the evaluation of complex expressions.

We produce a modified representation of the expression's source code using Clang's lexing and preprocessing systems. We iterate over each preprocessed token in the expression's source code. If the token was expanded from a user-defined macro then we add all of the expanded tokens to the modified representation. If the token was expanded from a macro defined in a system header, then we add the raw tokens covering the range the macro was expanded from. If the token was not expanded from a macro then we simply add it as-is. Tokens do not include newlines, so this method also fulfils our requirement of producing a single line of source code.

For an example of handling user-defined macros, consider Listing 2 (above right). The top-level expression is the initializer of `metres`: from the `2` to the final closing parenthesis. The tokens `2` and `*` are added to the modified representation as-is, because they do not involve macro expansion. The next token, `6372797`, is expanded from a macro defined in the user's source code, so we add the expanded tokens to the modified representation. All remaining tokens are added as-is, because they do not involve macro expansion.

Listing 2 User defined macro

```
#define EARTH_RADIUS_IN_METRES 6372797

double metres   = 2 * EARTH_RADIUS_IN_METRES
                  * asin(sqrt(x));
```

For an example of handling macros that are defined in system headers, consider the use of `S_ISREG` shown in Listing 3 (below). The top-level expression is the `if` statement's condition. The first token is expanded from a macro that was defined in a system header, so we find the area that the macro was expanded from and add the *raw* tokens to the modified representation: `S_ISREG(st.st_mode)`. The expanded tokens are discarded.

Listing 3 System macro expansion

Raw:

```
if (S_ISREG(st.st_mode)) {
```

Preprocessed:

```
if (((((st.st_mode)) & 0170000) == (0100000))) {
```

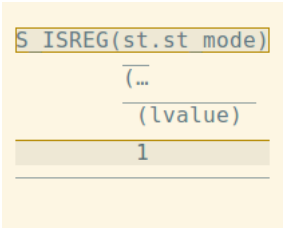

Figure 1: System macro evaluation

We annotate only the topmost expression from the *body* of expanded system macros in order to produce the "black box" representation discussed in Section 2. For example, consider the dynamic evaluation tree for Listing 3 shown in Figure 1 (above): the topmost node from the expanded body is shown (the `==` operator, with value `1`), and all other nodes from the expanded body are hidden (e.g. the `&` operator). We display nodes represented by the expanded *argument* to visualize the behaviour of the student's code.

The system next determines each expression's annotation text. SeeC provides information about the value produced by any expression's most recent evaluation. For example we will refer to the nodes in Figure 1. If the node's expression is a pointer or an lvalue then we use descriptive placeholder text for the annotation (e.g. the "`(lvalue)`"). For all other expressions we use SeeC's string representation of the value (e.g. the "`1`"). Annotation text that is too wide for the node is truncated, e.g. the node representing `st` is truncated from the full text "`(lvalue)`".

SeeC automatically detects several kinds of runtime errors during program execution, and provides information about detected errors during replay. We draw a dotted red line surrounding a statement's node if a runtime error was detected during that statement's execution, so that students may quickly locate

[4] http://clang.llvm.org

errors in the dynamic evaluation tree. Figure 2 shows the dynamic evaluation tree rendered when an invalid index is used as a subscript of `argv`.

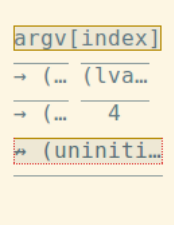

Figure 2: Statement with detected runtime error

The dynamic evaluation tree is a concise visualization of expression evaluation, but more information is available. To maintain clarity we use the "drill down" design, showing the following details in a tooltip when the mouse cursor hovers over an annotation:

- The complete annotation text.
- The expression's type. This allows students to observe the behaviour of type conversions (both implicit and explicit), and may be useful for debugging arithmetic errors (e.g. accidental use of integer division).
- A natural language explanation of the expression, as described by Heinsen Egan & McDonald (2014).
- A natural language description of any runtime errors that SeeC detected during the statement's execution.

Figure 3 shows an example of this tooltip. Further information and functionality is provided by integrating with, and deferring to, SeeC's other systems.

4 Integration with SeeC

The SeeC tool shows several complementary visualizations when replaying execution traces. We often reference these visualizations because the dynamic evaluation tree alone cannot conveniently represent all expression values, as we discussed in Section 2. In several situations we use placeholder text and direct students to other visualizations, e.g. to view an lvalue's designated object in memory.

Moving the cursor over a node in the dynamic evaluation tree causes its associated expression to be highlighted, in both the modified representation of the source code and the regular source code window. If the expression is an lvalue and has been evaluated, then its designated object will also be highlighted in the memory visualization window. Figure 3 shows both highlights: `lon2` is outlined in the source code window on the left, and `lon2`'s designated object is highlighted in the memory visualization on the right. If the expression is a pointer then the pointee object is also highlighted; this is necessary for observing rvalue pointers.

SeeC provides "*contextual navigation*" options, which we have also made accessible through the dynamic evaluation tree. Right clicking on any node provides navigation options based on the associated expression: move backward to the last time the expression was evaluated, or move forward to the next time the expression was evaluated. For lvalue expressions we also provide navigation based on the designated object's memory: move backward to its allocation, move forward to its deallocation, move backward to the prior time the memory was modified, or move forward to the next time the memory was modified.

5 Comparing visualizations

Our dynamic evaluation tree is not yet integrated with SeeC's source code window in the manner proposed by Lahtinen & Ahoniemi (2009): it occupies its own window within SeeC, in the manner of traditional expression evaluation visualizations. In this section we compare our implementation with existing visualizations, arguing that it offers several benefits despite not yet consolidating these windows. We will compare these visualizations with reference to Cognitive Load Theory as described by Sweller et al. (1998), and to the guidelines that Ware (2008) provides for information visualization based on current understandings of human perception and cognition.

Cognitive Load Theory provides guidelines for representing information to optimize intellectual performance and promote knowledge acquisition. These guidelines relate to optimizing the use of working memory: information must be in working memory in order to be processed, and working memory is extremely limited. Effective representations decrease *extraneous cognitive load*: the effect on working memory load of the manner in which information is presented, or of the activities required by students, i.e. that which is not intrinsic to the material at hand. Decreasing extraneous cognitive load enables students to devote more working memory to performing tasks and acquiring knowledge. This is particularly important when dealing with material that has a high *intrinsic cognitive load*. The *Split-Attention Effect* described by Sweller et al. (1998) is especially relevant to our comparison of visualizations. The Split-Attention Effect occurs when a student must mentally integrate two distinct sources of information in order to understand them, e.g. textual information that refers to a diagram, where neither the textual information nor the diagram are effective independently.

> On the basis of dozens of experiments under a wide variety of conditions, the evidence suggests overwhelmingly that it has negative consequences and should be eliminated wherever possible. (Sweller et al. 1998)

Ware (2008) provides a wealth of information concerning the effective design of information visualizations. Of particular relevance to program visualization systems are the recommendations on *optimizing the cognitive process*:

> The ideal cognitive loop involving a computer is to have it give you exactly the information you need when you need it. This means having only the most relevant information on screen at a given instant. It also means minimizing the cost of getting more information that is related to something already discovered. This is sometimes called *drilling down*. (Ware 2008)

There are two possibilities when attempting to get information related to something already discovered: either it is displayed somewhere else on the screen, or the user must perform some action to cause it to be displayed. Eye movements are much faster than mouse movements, but displaying too much information on screen will increase the difficulty of searching for any particular piece of information.

With this information in hand, let us now compare SeeC's implementation of dynamic evaluation trees

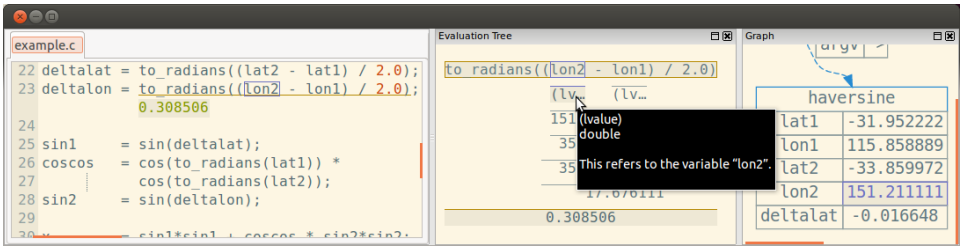

Figure 3: SeeC's highlighting and tooltip

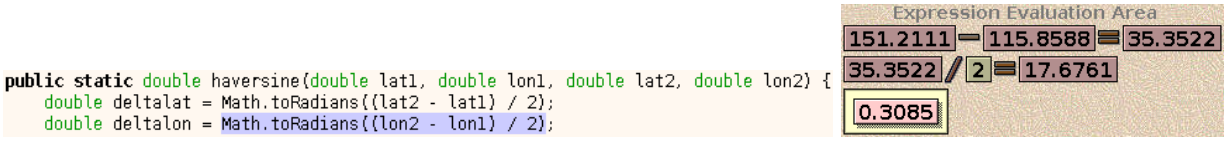

Figure 4: Jeliot 3 source code (left) and expression evaluation (right)

with the existing visualizations of expression evaluation used by novice focused programming tools.

Figure 4 shows a completed expression evaluation in Jeliot 3: operators and values are shown in the expression evaluation area, but students must consult the source code window for any other information about the expression. Thus the observed repeated switching of visual attention that motivated Lahtinen & Ahoniemi (2009) to propose the dynamic evaluation tree. This is a clear example of the Split-Attention Effect: the expression evaluation area alone is unintelligible, and students are forced to mentally integrate information from other windows in order to make sense of it. SeeC's dynamic evaluation tree, shown in Figure 5, contains a modified representation of the top-level expression's source code, so switching visual attention to the main source code window is only necessary when referring to other expressions or to the original representation.

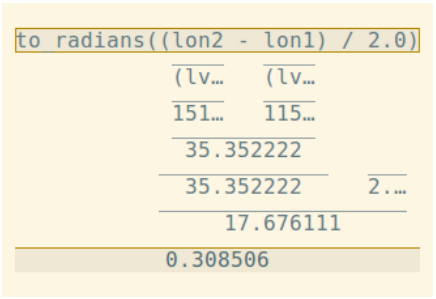

Figure 5: SeeC

The dynamic evaluation tree maintains a clear mapping between values and source code: the expression that produced a value occupies the same horizontal space as the value's node. Consider finding the expression that produced the value `35.3522` used in the division operation: in Jeliot 3 students must find the corresponding division operator in the source code window and then identify the left operand; in SeeC students can simply look at the top of the dynamic evaluation tree to see the source code occupying the same space as the value, or move their mouse cursor over the value to have that source code automatically highlighted. If a student wishes to determine *why* this expression produced this value using Jeliot 3 then they must find the correct subtraction operation in the evaluation history, perhaps by searching the right-hand side of the operations for the chosen value. Students using SeeC can simply look at the value's children in the dynamic evaluation tree.

SeeC consistently uses highlighting to visualize relationships and thus minimize the cost of finding related information, both within the dynamic evaluation tree and between different visualizations. We can see this highlighting in Figure 3 (above). The active expression is outlined in yellow in both the source code window and the dynamic evaluation tree. The annotation under the mouse cursor has its associated expression highlighted in violet, and as it is an *lvalue* the designated object is similarly highlighted in the memory visualization. This method is applied consistently throughout SeeC, e.g. moving the mouse cursor over an expression in the source code window will highlight the corresponding expression (and its produced value) in the dynamic evaluation tree.

In Jeliot 3, when a variable's value is used in an expression an animation shows the value "flying in" to the expression evaluation area. This provides only a transient association which, if it is important to the student's task, must be held in working memory, unnecessarily burdening their working memory load. Furthermore, the student may not know whether the association is important at the time the animation occurs, and there is no option to display the association after the fact: instead, students must determine the association themselves by mentally integrating information from Jeliot 3's multiple displays.

Bruce-Lockhart et al. (2007) described The Teaching Machine, a program visualization system supporting subsets of the Java and C++ languages, which also uses highlighting to illustrate relationships between different visualizations. Figure 7 provides an example: the active top-level expression's source code is highlighted in yellow, and the active sub-expression is an *lvalue* whose designated object (`lon1`) is also highlighted in yellow. If the student wishes to see the relationship between a different sub-expression and the values in memory, they must step backwards or forwards until that sub-expression is active.

The Teaching Machine visualizes expression evaluation using *expression rewriting*, in which an evaluated sub-expression's source code is replaced with its resulting value. Figure 6 shows the rewrite caused by an evaluation in The Teaching Machine: the underlined source code is the active sub-expression, which will be replaced by its result when the student steps forward. This visualization shows no history: students must step backwards to see previous operations. Furthermore, an operation's operands and result are

not simultaneously visible, so considering an operation requires a student to hold relevant information in working memory while stepping forwards or backwards. Effectively, the student is required to mentally integrate information from two visualizations which cannot be displayed simultaneously. The dynamic evaluation tree does not require this information to be held in working memory, because it is always accessible via rapid eye movements or mouse hovering.

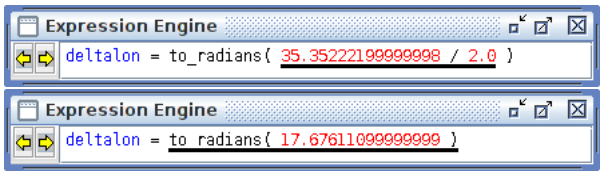

Figure 6: The Teaching Machine 2's rewriting

Expression rewriting is also used by WADEIn, a web-based tool designed to help students construct knowledge of C's expression evaluation rules, presented by Brusilovsky & Spring (2004). WADEIn annotates the source code of an expression with the order in which the individual sub-expressions will be evaluated (shown as numbers beneath the sub-expressions). The evaluation of the complete expression is visualized by a "shrinking copy" of the source code: the active sub-expression is copied into an "evaluation area", its evaluation is visualized, and the result then replaces the original sub-expression in the "shrinking copy". Only the active sub-expression's evaluation is shown, so students must step backwards and forwards to observe the evaluation of different sub-expressions. WADEIn is a tutoring system for isolated expressions: it supports only mathematical and logical operators with `int` and `double` type variables. The system tracks the student's exposure to different operators, increasing the speed of animation and removing certain sub-steps as the student's "level of knowledge" increases.

The dynamic evaluation tree is the only method of visualization that shows every step of a complex expression's evaluation in a single image while maintaining relationships from evaluated sub-expressions to their original source code, and from lvalue expressions to their designated objects. Considering the advice and information provided by Sweller et al. (1998) and by Ware (2008), we believe the dynamic evaluation tree is a significant advancement in terms of both reducing extraneous cognitive load and optimizing the process of finding information that is related to something already discovered.

6 Limitations and future work

Future developments should be guided by the requirements of novice programmers learning the C programming language, thus the most important remaining task is to evaluate the dynamic evaluation tree's usage by novice programmers. We are currently investigating SeeC's usage by students in our second year course covering the C programming language and Operating Systems, and will be collecting feedback from students including their perceptions of the dynamic evaluation tree visualization and their suggestions for future development. During our own development and use of the dynamic evaluation tree we have identified some potential areas of investigation, which we describe in the remainder of this section.

We use Clang's expressions to generate our dynamic evaluation tree. This reduces our system's implementation requirements and provides robust, complete support for the C programming language, but could expose technical details that may confuse novice programmers. We hide some information from students: for example, in Figure 5, we hide the expressions representing the reference to `to_radians` and its decay to a function pointer. It may be useful to provide an option to display all expressions, or to implement an adaptive system that reveals technical details when a student's knowledge is sufficiently advanced.

User-controlled information eliding may also be useful for handling macro expansion. Our implementation either fully expands or does not expand macros, but in some situations it may be useful to show a *partial expansion*. Listing 4 shows a definition for the function-like macro `S_ISREG`; a raw use of this macro; and a partial expansion of this use, in which the expanded tokens have not undergone *rescanning* which would have expanded `S_IFMT` and `S_IFREG`. Showing `S_IFMT` and `S_IFREG` rather than their expanded numeric constants may be more informative than the fully preprocessed code (e.g. shown in Listing 3). Students could interactively control whether individual macros are expanded, allowing them to inspect the preprocessor's actions and to select an appropriate representation for the task at hand.

Listing 4 Partial macro expansion

Macro definition:
```
S_ISREG(m) (((m) & S_IFMT) == S_IFREG)
```
Raw:
```
S_ISREG(st.st_mode)
```
Partially expanded:
```
(((st.st_mode) & S_IFMT) == S_IFREG)
```

The dynamic evaluation tree visualizes the values produced by each expression, but it does not represent expressions' side effects. For example, a postfix increment operator's node would show the value loaded from the operand's designated object, but would not indicate that the object's value was modified. This problem is generalised by annotating function calls, which may have numerous side effects. It may be useful to visually indicate that an annotation's associated expression caused some side effects. The exact nature of the side effects could be represented in the tooltip produced by hovering the mouse cursor over the annotation.

7 Summary

The dynamic evaluation tree concisely visualizes expression evaluation while maintaining a visual relationship between each expression's source code and its produced value. The complete history of a complex expression evaluation can be shown in a single static frame, enabling students to rapidly scan each step of the evaluation. In this paper we generalised the dynamic evaluation tree to account for arbitrary expressions in the C programming language, presented our implementation of the dynamic evaluation tree for the novice-focused program visualization and debugging tool SeeC, and compared this implementation to previous visualizations of expression evaluation.

We believe that the complicating factors discussed and mitigated within this work will support attempts to implement the dynamic evaluation tree in other novice-focused tools, regardless of their supported programming languages. For example, the difficulties of representing pointers may also apply to the representation of references in Java or Python.

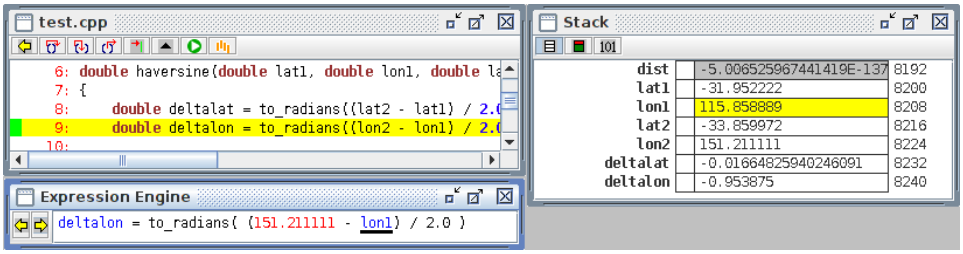

Figure 7: The Teaching Machine 2's highlighting

The dynamic evaluation tree was introduced by Lahtinen & Ahoniemi (2009) with the intention of reducing novice programmers' switching of visual attention while using program visualization tools. To our knowledge, we have presented the first implementation of this concept. We believe this is a robust, maintainable implementation and yet its development was straightforward, which speaks to the underlying SeeC system's potential as a foundation for novice-focused program visualization research.

Finally, this implementation enables investigation of the dynamic evaluation tree's usefulness for novice programmers learning the C programming language.

8 Acknowledgements

This research is partially supported by an Australian Postgraduate Award.

References

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K. & Zander, C. (2007), 'Threshold concepts in computer science: Do they exist and are they useful?', *SIGCSE Bull.* **39**(1), 504–508.

Bruce-Lockhart, M., Norvell, T. S. & Cotronis, Y. (2007), 'Program and algorithm visualization in engineering and physics', *Electron. Notes Theor. Comput. Sci.* **178**, 111–119.

Brusilovsky, P. & Spring, M. (2004), Adaptive, Engaging, and Explanatory Visualization in a C Programming Course, *in* 'ED-MEDIA'2004 - World Conference on Educational Multimedia, Hypermedia and Telecommunications', pp. 21–26.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L. & Zander, C. (2008), 'Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers', *Computer Science Education* **18**(2), 93–116.

Heinsen Egan, M. & McDonald, C. (2014), Program visualization and explanation for novice C programmers, *in* 'Sixteenth Australasian Computing Education Conference (ACE 2014)', Vol. 148 of *CRPIT*, ACS, Auckland, New Zealand, pp. 51–57.

Lahtinen, E. & Ahoniemi, T. (2009), 'Dynamic evaluation tree for presenting expression evaluations visually', *Electronic Notes in Theoretical Computer Science* **224**, 41 – 46. Proceedings of the Fifth Program Visualization Workshop (PVW 2008).

Moreno, A., Myller, N., Sutinen, E. & Ben-Ari, M. (2004), Visualizing programs with Jeliot 3, *in* 'AVI '04: Proceedings of the Working Conference on Advanced Visual Interfaces', ACM, New York, NY, USA, pp. 373–376.

Rountree, J. & Rountree, N. (2009), Issues regarding threshold concepts in computer science, *in* 'Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95', ACE '09, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 139–146.

Sweller, J., van Merrienboer, J. & Paas, F. (1998), 'Cognitive architecture and instructional design', *Educational Psychology Review* **10**(3), 251–296.

Ware, C. (2008), *Visual Thinking for Design*, Morgan Kaufmann Publishers, 30 Corporate Drive, Suite 400, Burlington, MA, USA.