GRAPHICAL USER INTERFACE FOR PROTRACE LIBRARY

Konrad Grzanek

IT Institute, Academy of Management, Lodz, Poland
kgrzanek@swspiz.pl

Abstract

Protrace library allows Clojure programmers to investigate an arbitrary computational process at the abstract syntax tree (AST) level. Abandoning pure textual representations and moving towards graphs and trees increased the readability and made the insight into processes easier. It gains special importance when tracing recursive and mutually dependent procedures execution. Prefuse visualization framework provided great features to prepare convenient views of extended ASTs. The paper presents recent Protrace improvements in this matter.

Key words: Protrace, Clojure, Prefuse, Functional Programming, Computational Process, Visualization

1 Introduction

Protrace expression tracing library developed recently and described in [1] allows Lisp programmers to investigate an arbitrary computational process at the abstract syntax tree (AST) level. According to [2] "the concrete syntax definition is derived from the abstract one and that the abstract syntax definition is the true definition of language structure". This is why the abstract syntax trees are such a convenient form of presenting language constructs that appear in the source code of programs.

The library was created to trace programs' dynamics in the first place, not only the structural dependencies. This is why a decision was made to use a custom extended version of ASTs. The extension involves adding subsequent AST nodes to the whole tree structure visualizing a flow of substitutions (in the computational model based on substitutions as described in [3]) that take place during expression(s) evaluation. It is to be presented in a more expressive way at the following figures.

Previous version of Protrace used textual AST representations [1]. Unfortunately, the nature of most non trivial computations is such that it requires many substitutional steps. Displaying the textual representation of the whole extended AST on the console leads to a hardly readable form, very inconvenient to use when looking for bugs or when searching for some behavioral patterns in the analyzed software. This is why we decided to use a visualization framework [6] to implement a more convenient graphical AST view.

It is also worth mentioning that all implementation activities described in this paper were applied to a new Clojure [4, 5] version of Protrace.

2 Prefuse visualization framework

Moving Protrace library into Java platform (Clojure is a JVM language) was a major step that allowed us to use a whole Java libraries stack. When choosing the visualization framework, we searched for the following features:

- specialization towards visualizing graphs, trees in particular,
- simple and readable, yet attractive appearance,
- convenient Java API,
- Java-pureness, the lack of dependencies on custom platform-specific libraries to ensure conformity with WORA (write once run anywhere) principle of the Java platform

A choice was made to use Prefuse [6, 7] visualization framework. This Java library offers a whole bunch of possible graph displays and associated algorithms:

- tree display (actually used to visualize Protrace extended ASTs)
- tree-map display
- various graph visualizations, including hyperbolic and fisheye distortions, forces, simplifications, birdviews.

It also offers a pretty consistent API and allows very deep customizations. The framework is not written in the functional style due to its purposes and high performance requirements, so we proposed and implemented a simple integration layer for Clojure and Prefuse. There were also some bugs inside the Prefuse engine resulting in subtle in nature, yet prominent in size memory leaks (loitering objects problem) requiring fixes.

3 Usage

To show a trace of an arbitrary Clojure s-expression evaluation one must take care of two things:

1. Wrapping selected expression with a ***(watch** ...)* form
2. Running the wrapped expression inside an environment created by ***(show-trace** ...)* syntactic form.

For example, if someone wants to trace an expression:

```
(+ 1 2 (* 3 4 (- 6 7))
```

at all levels of the computational process, has to beused the following form:

```
(show-trace (watch-all (+ 1 2 (* 3 4 (- 6 7))))
```

The interior ***(watch-all** ...)* form wraps every sub-expression with a *(watch...)* form. The usage and nature of ***(watch-all** ...)* is to be presented in a separate paper.

4 Extended Abstract Syntax View

The following

Figure 1 shows the main visualization window after executing the above ***(show-trace** ...)* form.

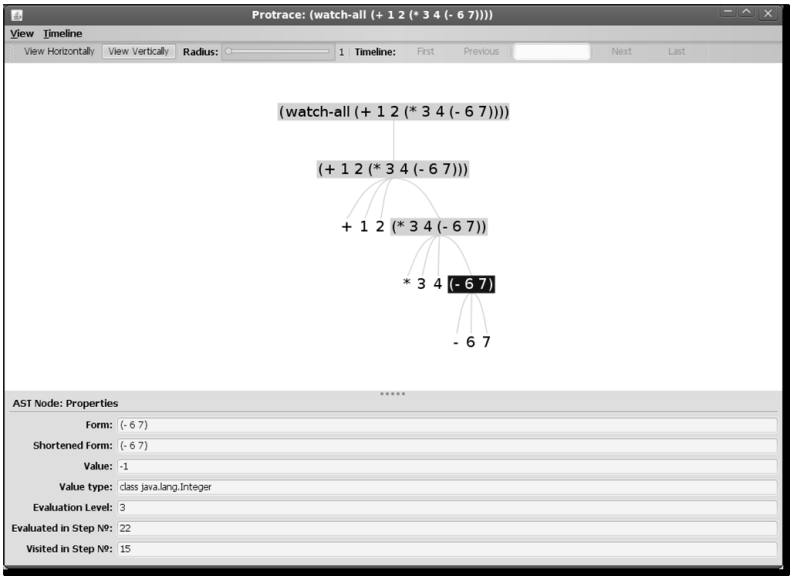

Figure 1. Extended AST View

Protrace GUI consists of a central panel showing an extended AST view and a properties panel below. The selected node is marked with a blue color and the path in the tree leading from the root to the selected node is marked with pale-blue. The tree is always expanded into a depth specified by the radius below the selected node. Selecting an AST node causes the properties panel to display important properties of the selected node:

- **Form:** the syntactic form of the s-expression represented by the node
- **Shortened form**: the syntactic form cut into a predefined length for textual presentation purposes
- **Value**: of the s-expression and its
- **Value type**: resolved using a standard Clojure type info routines
- **Evaluation level**: the node level in the extended AST tree. The tree root's level is 0.
- **Evaluated in Step**: a number of a step of the computational process when the node was assigned a value by the (meta-cyclic) evaluator
- **Visited in Step**: a number of a step of the computational process when the node was visited by the evaluator

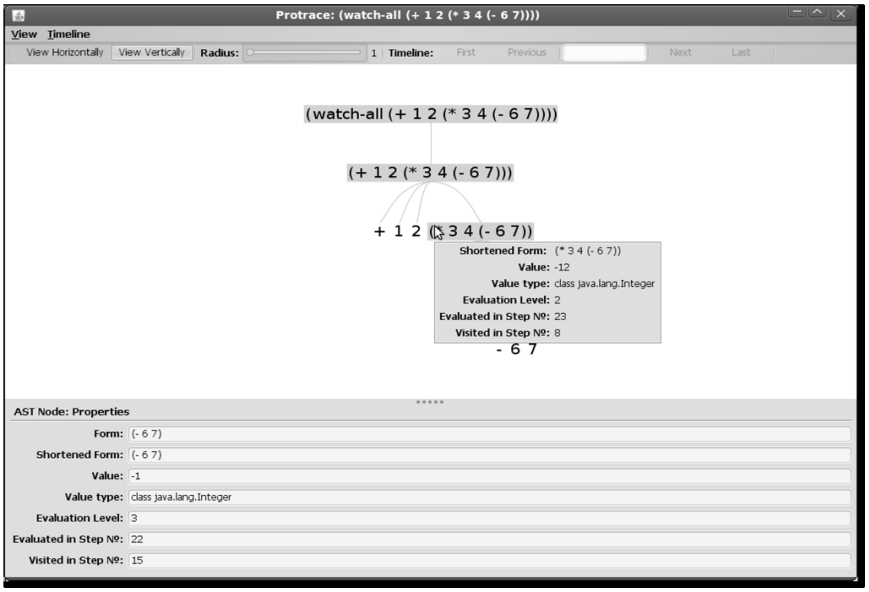

Figure 2. Tool tips in the extended AST View

Additionally the AST view offers tool tips for nodes pointed by the mouse to perform fast overview of the properties without changing the overall tree layout and properties of the panel's state. This feature is presented above in Figure 2.

5 Recursive procedures and their evaluation traces

Recursion is one of the key techniques of functional programming. Protrace is highly effective when analyzing this kind of looping. To analyze a *factorial* procedure like the one below:

```
(defn silnia [n]
   (if (zero? n) 1 (* n (silnia (dec n)))))
```

one must wrap the procedure body with the mentioned ***(watch-all** ...)* form or selectively with ***(watch** ...)* on the specific sub-expressions:

```
(defn silnia [n]
   (watch-all (if (zero? n) 1 (* n (silnia (dec n))))))
```

Now executing the form: (show-trace (silnia 3)) in the Clojure REPL causes the system to display a window with an extended AST like in Figure 3.

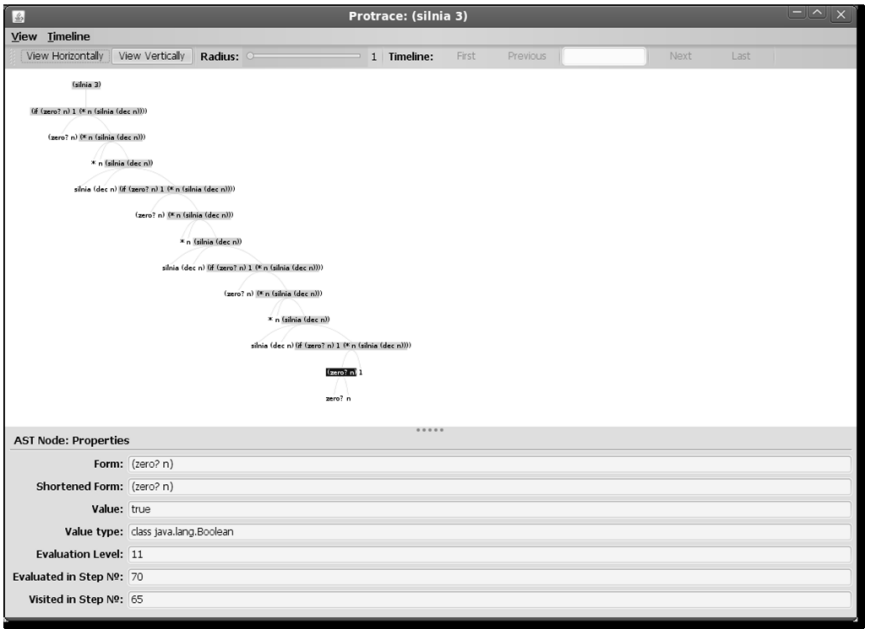

Figure 3. Recursive function execution trace in the extended AST View

When the evaluation leads to a recursive behavior, the AST contains node duplicates on various levels. Every expression instance holds a bound value at its evaluation level. This allows one to trace the execution of complex procedures, searching for the stop conditions etc.

The view may be reorganized to be displayed both vertically and horizontally. See Figure 4There is also a radius slider in the GUI toolbar. Tree radius may change from 1 to 7. Setting a higher radius value causes the AST tree to be expanded to a greater depth, thus allowing to make a more complex overview of the traced process. The impact of the radius value may be observed when comparing Figure 4 and Figure 5.

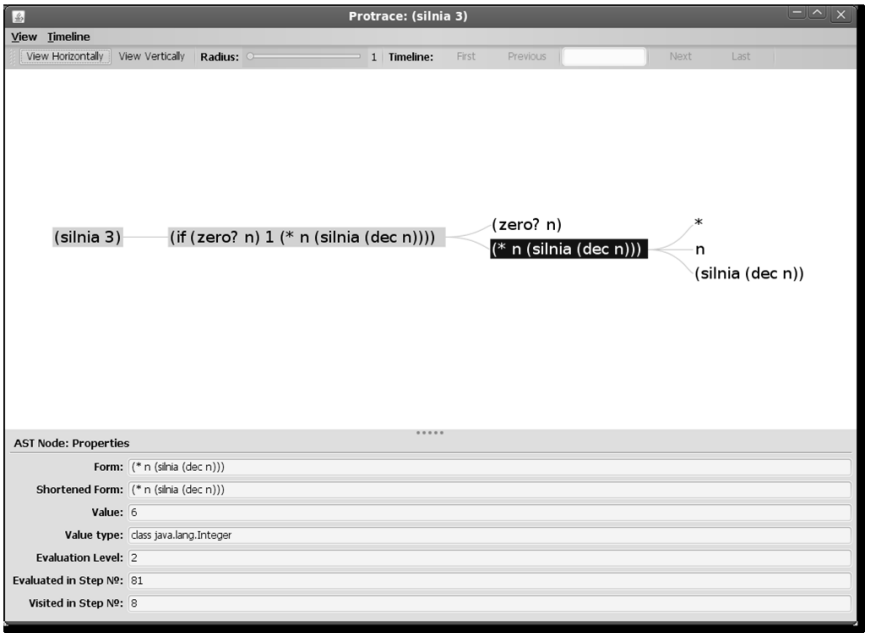

Figure 4. Horizontal orientation of the extended AST view. Tree radius set to 1.

6 Summary

With Prefuse visualization framework Protrace gained a convenient way to observe the traced computational process at different stages of evaluation. Abandoning pure textual representations and moving towards graphs and trees increases the readability and makes the insight easier. It is especially important when tracing recursive and mutually dependent procedures execution. There are plans to extend the GUI with a mechanism of (semi)automatic stepping across the extended AST nodes to follow the changes in the evaluator taking place on different stages of the computational process. Additionally, the AST nodes would be marked with different colors underlining their position in the process in the overall timeline.

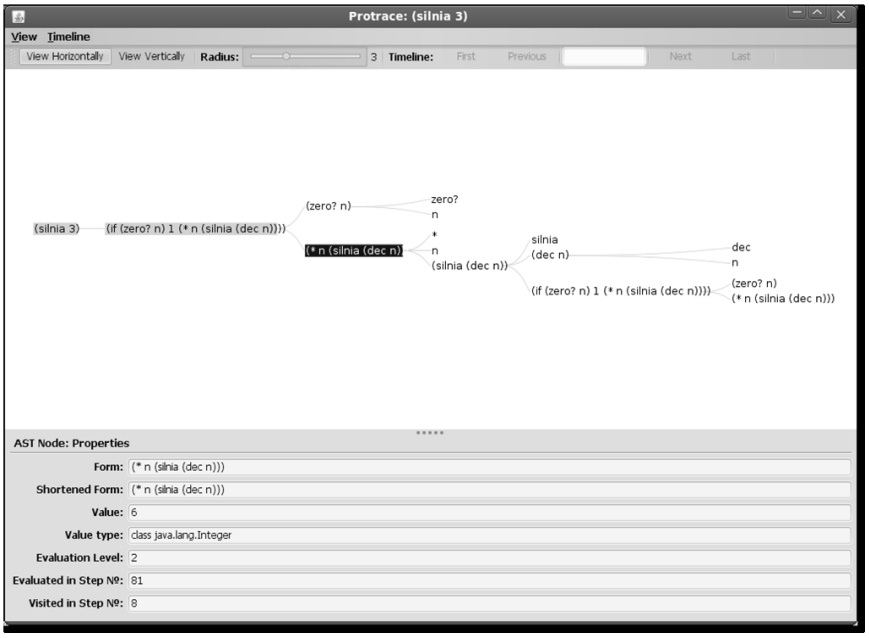

Figure 5. Radius (value 3) impact on the extended AST view

References

1. Grzanek K., Cader A., 2010, *Protrace: effective recursion tracing and debugging library for functional programming style in Common Lisp*, Artificial Intelligence and Soft Computing, 10Th International Conference, ICAISC 2010, Zakopane, Poland, June 13-17, 2010, Part II. Lecture Notes in Computer Science 6114 Springer 2010, ISBN 978-3-642-13231-5, pp. 468-475
2. Schmidt D.A., 1986, *Denotational semantics: a methodology for language development*, Allyn and Bacon, ISBN 0205104509
3. Abelson, H., Sussman, G. J., 1984, *Structure and Interpretation of Computer Programs*, MIT Press, ISBN 0-262-01077-1
4. Halloway S., 2009, *Programming Clojure*, ISBN: 978-1-93435-633-3, The Pragmatic Bookshelf.
5. Clojure Website, November 2010, http://clojure.org
6. Prefuse Website, November 2010, http://prefuse.org
7. Heer J., Card S.K., Landay J.A., 2005, Prefuse: *A Toolkit for Interactive Information Visualization*. Proc. ACM Human Factors in Computing Systems (CHI), pp. 421-430.