GUI Environments for Functional Languages

Daniel J. Pless & George F. Luger
University of New Mexico, Computer Science
Ferris Engineering Center
Albuquerque, NM 87131
(505) 277-9424

{dpless,luger}@cs.unm.edu

Submitted to the Onward track of OOPSLA-03

ABSTRACT

We describe an Integrated Development Environment (IDE) for purely functional programming languages. This environment leverages the properties of such languages to provide software development productivity features that would be difficult or impossible to provide for a language with state. We show how such an environment can enable the construction of regression tests and improve documentation. We also show how other standard tools such as debuggers and profilers can leverage these capabilities.

Keywords

GUI Environments, Functional Programming

1. INTRODUCTION

1.1 Motivation for Functional Languages

There has been extensive research in pure functional programming languages [2,4,6-8]. These languages are characterized by the property that there is no "state" in the language. That is, no variable changes its value after its initial assignment. Functions always return the same value given the same input, with no side effects. The advantage of a stateless language is that functions are completely characterized by their input/output behavior.

An important result of a purely functional language is its support for lazy evaluation, which greatly simplifies the specification of control. This allows the principled construction and manipulation of large or countably infinite structures. Hughes [4] points out that lazy evaluation is a powerful tool for modularity. In addition, these functional languages allow automatic analysis of code including a powerful polymorphic type system and the possibility of automating parallelization. In this paper, we argue that purely functional languages also can support useful software engineering features in a GUI environment.

1.2 Aim of This Paper

A standard method to improve programmer productivity is to design an Integrated Development Environment for a language. In some cases, such as Smalltalk, the environment is designed specifically for the language. In Smalltalk, each object maps to a window in the environment. The environment takes advantage of the structure of the language to effectively present an interface for the developer.

A more mainstream approach to IDE design is to see the development environment as a programmer-centric tool, independent of the features of a particular language. An advantage of this approach is that the environment can be generic and applied to different languages. Microsoft's Visual Studio is built around this premise, with the same development environment for many different languages. We believe that the properties specific to purely functional languages demand designing a GUI environment to take advantage of these characteristics. We believe that such an environment will enhance the popularity of functional languages and in turn greatly improve programmer productivity.

In this paper, we propose the design of a visual development tool that takes advantage of the structure of purely functional programming languages. We have chosen to focus on Haskell [5] because it is one of the most widely known purely functional programming languages. Most of the ideas in this paper can be applied to other purely functional languages such as Miranda [8], Clean [2], or Mercury [7]. To keep the exposition simple and coherent, we focus only on Haskell.

There already exists an IDE for one functional language, namely Clean. The CleanIDE implements many features that exist in other development environments. However, with the exception of a theorem prover, it does not include features specifically tailored to purely functional languages. Our aim in this paper is to describe what these features should be.

1.3 Outline of This Paper

This paper continues as follows: Section 2 presents an overview of our proposed Integrated Development Environment. Section 3 describes in more detail the various components of the system that support writing code. Section 4 describes how other tools such as a Graphical User Interface (GUI) builder or code versioning can be integrated into the system. Finally, Section 5 gives some concluding thoughts.

2. OVERVIEW OF ARCHITECTURE

The system will consist of a GUI environment with a number of classes of windows similar to other development systems. Figure 1 shows the different classes of windows and how they relate. Arrows in the figure indicate which windows can invoke or change other windows. The windows grouped together in the bold rectangle represent the core of the system for writing code, which is described in Section 3. Outside this rectangle are peripheral tools for software engineering as described in Section 4.

The user will work with editor windows containing source code from different files. Interpreter windows associated with the editor windows provide the means for the developer to quickly test functions. The interpreter will also capture information useful for

constructing regression tests and for documentation, as is explained below.

The user can open debugger windows on any expression evaluated in the interpreter windows. The debugger will show current evaluation location in the editor. The user can open object viewer windows to explore large structures from either the debugger or editor windows. The developer will also have tools for code versioning, profiling, GUI building and a theorem prover.

Generally IDEs include project management and automatic build tools as well. We will not describe any in this paper, as such tools will not be very different for functional languages than they are for any other type of language.

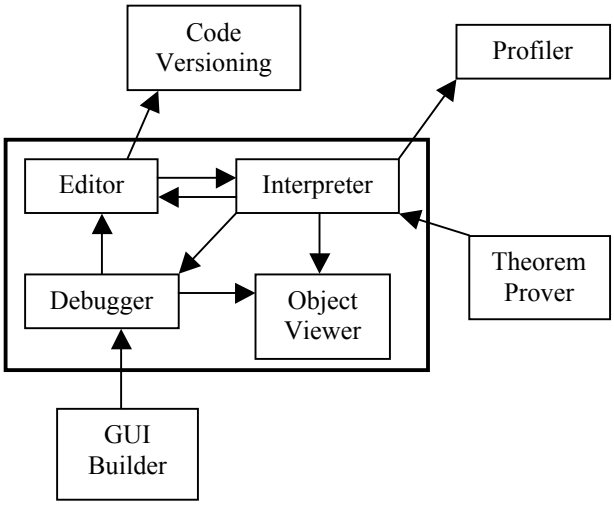

Figure 1. A diagram of components of the system. The components in the bold rectangle represent the core of the system as explained in Section 3.

3. MAJOR COMPONENTS

Following up on the architectural description of the previous section, we now describe the major components of the system. These include the editor, interpreter, debugger, and object viewer. In Section 4, we will survey other features that could enhance the IDE.

3.1 Editor

The editor window will have context-sensitive syntax coloring. The user can instruct the editor to add or display type information that the compiler inferred about top level declarations. Even thought many Haskell programmers explicitly use type signatures, this editor will automatically generate them when possible. The user may wish to display them for annotating functions, or hide them to increase visible code density.

When the user compiles, the system will try to help correct errors. On a compile time error, the expression containing the error will be highlighted in the editor window. A dialog box will pop up with an error message. The box will also contain suggestions for fixing the error, similar to a spelling or grammar checker in a word processor. A standard spell-checking algorithm will be used to suggest corrections for an undefined identifier. The identifiers that are visible in scope will be used for the "dictionary" for correcting this type of error. For other errors, a variety of heuristics will be used. If changing the order of arguments repairs a type mismatch error, that will be a suggestion. For mismatched parentheses, the system might search for an insertion or deletion of a parenthesis that is type-correct. The system will not be able to suggest correction in all situations, just as in the word processing case. It may also make erroneous suggestions.

3.2 Interpreter

It is quite common for a development environment to provide an interpreter for testing out code. Such an interpreter can be used to quickly test bits of code to see if they work. However, there is other information that can be captured from such work that other interpreters will fail to capture. The expressions that the user types for newly written code can be reused for documentation and regression testing purposes as described later in this section.

Our interpreter does not contain an ordered sequence of command lines as a normal interpreter would. Instead, it contains a set of expression cells, similar to Mathematica or Maple. Each cell will contain an expression queried by the user and the resulting evaluation (with type information). Such cells will be editable and movable. This is possible for Haskell because the language is purely functional; evaluating an expression can produce no side effects on the system. Thus, the user can reorder the expression cells without changing the correctness of what is displayed. If the user edits the expression, the results would be similarly updated.

The expression cells will be displayed as a scrollable list separated by thin or dotted lines. The user can click on a cell to select or edit. There will always be a blank cell at the end of the list for the user to create new expressions. If there is an error in the expression that the user types, the same correction suggestion algorithm used for helping correct compiling errors will be run. Thus, the user will see a dialog box explaining the error with clickable suggestions for fixing it. If a runtime error occurs, the user will have the option of opening a debugger window.

The expression cells will be selectable for drag and drop moving as well as for deleting. It will also be possible for the user to pop-up a menu for any cell. This menu will contain a number of cell maintenance functions. One function will find and highlight the source code for the function called in the cell. Another will start a debugger window on the exact expression being evaluated (the debugger is described in Section 3.3).

There will also be menu items for sorting the cells in various ways. There will be a function for sorting the cells by the lexicographical order of the expressions. It will also be possible to sort by the order in which the cells were first typed. It is also useful to sort based on the computation time for evaluation. Finally, the user will be able to partition the list based on whether the cell is part of a regression test (described next).

An important pop-up menu function is the ability to lock the result of a cell's evaluation into a regression test. The user will do this when a function is producing the correct result on some input, and the user wishes to ensure that it remains correct. If, because of changes in the code, the value in some regression tested cell changes, the system will flag this as an error. A message box will appear giving the user the options of running the expression in the

debugger, updating the regression test to indicate that the new value is now correct, or disabling or deleting the test. A value must be a member of Haskell class EQ to be made into a regression test. This is necessary in order that the Haskell system can test whether the value has changed. Requiring regression test values be members of class EQ isn't a major restriction as almost all values that a user wishes to inspect are members of that class.

The objective of constructing regression tests this way is to change the process by which they are built. Instead of building the test entirely after building a large chunk of code such as a module, one builds regression tests incrementally. Write a little piece of code, test it, and lock in the results. As one builds up the code, the earlier tests can help prevent errors from creeping in later. Building these tests requires little additional effort by the programmer, as they come from the work in the interpreter that she is already doing.

An interpreter window will also contain a small scrollable list of open modules. The purpose of this is to allow the user to define test code and test objects in a file other than the source code. One does not open modules through a command in an interpreter cell, as this would destroy the state-free semantics of the interpreter.

All of the information in an interpreter window will be saved to a file. The file will have the same name as the source file with which it is associated, but will have a different extension. The idea is that the information in the interpreter should be captured, not lost at the end of a session. One purpose of this capture is for regression testing as indicated above.

However, the information captured can also be a form of documentation. Often, the best explanation of the purpose and usage of a function is examples. However, it requires effort to construct such examples and it is difficult to know if these examples are up to date and reflect the current operation of the code. Our system will automatically capture such information from the users natural testing and debugging cycle, and will update such cells on each compile. The sorting and delete options described above will aid the programmer in managing and reviewing these examples.

3.3 Debugger

The debugger is usually started by the user invoking a pop-up menu on an interpreter cell. There are three reasons for designing the interface to the debugger this way, rather than having the user first "turn on" debugging and then type an expression to be debugged. First, this design makes the interface less modal. Instead of being in a whole new state when debugging, the debugger simply is another window with which the user can work. Therefore, the user will be able to open multiple debugger windows on different expressions. These debuggers will not interfere with each other, as a Haskell function cannot produce side effects.

The second reason for debugging already existing and evaluated expressions is that it supports a natural workflow. One does not start up a debugger to see if something goes wrong. One normally will try something in the interpreter, see that the resulting answer is incorrect, and then start the debugger. Our system directly supports this more natural order of actions.

The third reason for directly debugging evaluated expressions is to support interpreter cells as a form of documentation. As we noted in Section 3.2, it is useful to see examples of using some function. However, by stepping through an example in the debugger, the user can also see how the function works. In most systems, it is too much trouble to set up an example in the debugger to be worth doing. But in a system where getting an example started was extremely quick, the user would get a good idea of how some code works by stepping through it on a few examples.

When the debugger is started on an expression, it presents the user with a debugger window stopped at initiation of evaluation. If the debugger was started as the result of a runtime error, it will be stopped at the point of the error. The user will then have the standard functions available for debugging. The debugger will highlight in the editor window the next expression to evaluate. The user will be able to step into the next function call, step over it, or step out of the current function call altogether. The debugger will display the current call stack and the values in the currently displayed scope. The user will be able to move to any level in the call stack, as well as step over or out of the evaluation at any level. The debugger will display the value bindings for the local scope in the call stack at any level.

3.4 Object Viewer

Usually, values are simply printed when needed in the debugger or interpreter. There are times when values are too complex to be displayed, either by the debugger or the interpreter. A common way to handle this problem is to display as much of the object as is feasible, using an ellipsis to indicate fields that cannot be displayed in a small space. Our system will allow the user to click on such an ellipsis to view the field in another window. This window will show a breakdown of the object and will allow one to burrow into complex structures, similar to other IDEs.

One big difference in our object viewer is that since all values are immutable, the user will not be able to edit them. In general, values displayed from the debugger will not change as one steps through the code. The exception to this is values that contain unevaluated parts (thunks) as a result of lazy evaluation. As these unevaluated pieces are partially or fully evaluated, the window will be updated.

4. OTHER POSSIBLE FEATURES

One can imagine other tools for this system; we give four possibilities in this section. The common thread among these suggestions is that they all take advantage of the state-free nature of functional programming or build on features that do.

4.1 GUI Builder

The tool described so far is useful for developing the logic of an application. However, it doesn't provide much support for quickly developing the advanced interfaces that users demand. We propose that a full-fledged GUI builder is needed for handling this. The builder will allow developers to layout windows and dialog boxes with a palette of controls. This can be similar to other GUI builders for languages such as Java or Visual Basic. The difficult issue for a purely functional language based GUI builder is how to deal with event functions.

In a GUI builder for a language with state, the system responds to events by employing user-defined callbacks, known as event handlers. These functions generally work by performing side effects. For example, a callback function might change a value in

a field of some form. In our system, the user will instead specify what "state" is to change and the function call to determine the new "state". This information will be wrapped in a monad indicating what monadic binding is to be invoked. The basic idea in our approach is to have the application framework handle the "mutable" data. That is, it will be in charge of sequencing the monads; the user's task will be to write functions that describe the transformation of state. The system then becomes the repository of mutable data, and the programming remains purely functional.

One advantage of this system is that the programmer can generate regression tests for event handlers. This is difficult for other systems except as part of a larger system test. In our system one can set a breakpoint on an event function and use the debugger to capture the input to the handler for a regression test. This way, correct behavior for something as small as an event function can be locked in.

4.2 Profiler

The system will support code profiling as well. In particular, it will allow the user to profile all the runs in a project's regression test. This could be used by the user to see performance bottlenecks; it also could be used to check code coverage of the regression test. This information could also be provided to the compiler to do more intelligent optimization [3]. The advantage that functional programming provides in this situation is that it is easy to construct the regression test and thus one always has a number of cases for use by the profiler and optimizer.

4.3 Code Versioning System

A code versioning system will be coupled with the environment. If a change in a function caused a regression test to fail, the code trace could be compared with the code trace of the previous version. In a purely functional language, it should be possible in principle to automatically track down which changed function caused the regression test failure. The versioning system might be used by multiple developers on a large project. It can be quite difficult to discover responsibility for a system test failure, say on a nightly build. Having a tool to compare version traces and automatically locate the function or module where changes cause the error will be quite useful.

4.4 Theorem Prover

The language Clean now comes with the theorem prover "Sparkle" [5]. Purely functional languages, by their nature, are extremely well suited for the automated manipulation that is required for proving properties of code. A theorem prover could be added to the regression testing system. The user would then assert properties for the prover to attempt to verify. A simple example is asserting that a function that reverses a list is its own inverse. That is, reversing a list twice returns the original. Once such an assertion is proven, the proof could be added to the regression test. If changing the code results in breaking some expected property, the user would be warned. In some sense, the regression tests that can be captured by the interpreter are a special case of the general properties that can be specified to a prover.

5. FINAL COMMENTS

5.1 Outstanding issues

We have presented an architecture for Integrated Development Environments for purely functional languages. There are a number of issues that remain to be considered. One possible problem is that regression testing may slow the compile cycle too much. In the worst case, a regression test may simply fail to halt. There are two means of handling this issue. First, the user will be able to indicate that certain tests are to be run only when a full regression test is explicitly requested. The second method is to implement a user adjustable computation threshold that will halt long computations. Regression tests that exceed the computation thresholds could be reported as errors or warnings.

Another issue is how to handle the I/O monad in regression testing. The problem is that the results of such code can depend on the state of the world. The easiest way to handle this is to not allow the direct use of the I/O monad in the interpreter window. Testing such code will require running the entire program.

As noted in Section 3.3, the functional properties of Haskell allow multiple debugging in parallel. Multiple debugger windows cannot interfere with each other except in their effect on the editor window. That is, each one will want to select a different range of text in the editor window. This is not a major problem if each time a debugger stops, it changes the selection range. Thus, only the last debugging step requested by the user will be displayed in the editor.

One last issue is the display of unevaluated thunks in an object viewer. The system should allow the user to see an indication of whether or not the function in the thunk is likewise an unevaluated object. In addition, the user will see the arguments to this function.

5.2 Conclusions

Finally, consider what developing in this environment might be like. You type parts of your code into a context sensitive color-coded editor window. When you compile, the system suggests corrections for your errors using the type system to guide its suggestions. After you compile, you use expression cells to test the code's operation. If you find that one or more tries are incorrect, a debugger window for any or all of them is a pop-up menu away. When you make changes to the code, the expressions automatically reevaluate, without needing to be retyped. As you find expressions that do evaluate correctly, locking this correct behavior into a regression test is also pop-up menu away. When you are done debugging this portion of your code, you already have a searchable/sortable set of examples using these recently created functions. Any subset of these expressions can be registered into a regression test.

6. ACKNOWLEDGEMENTS

Our thanks to Hajime Inoue for reading and critiquing an early draft of this paper and to Harry Chomsky for useful discussions on tools for functional programming.

7. REFERENCES

[1] Achten, P. and Peyton Jones, S. Porting the Clean Object I/O library to Haskell. *Proceedings of the 12th International workshop on the Implementation of Functional Languages,* 2000, 194-213.

[2] Brus, T., M.C.J.D. van Eekelen, M. ,van Leer, M., and Plasmeijer, M. CLEAN - A Language for functional graph rewriting, *Conference on Functional Programming Languages and Computer Architecture*, 1987, pp. 364-384.

[3] Chang, P., Mahlke, S., and Hwu, W. Using profile information to assist classic code optimizations. *Software - Practice and Experience*, 1991, 21(12), 1301-1321.

[4] Hughes, J. Why functional programming matters. *Computer Journal*, 1989. 33(2), 98-107.

[5] de Mol, M., van Eekelen, M., and Plasmeijer, R. Theorem Proving for Functional Programmers - SPARKLE: A functional theorem prover. *Proceedings of the 13th International Workshop on the Implementation of Functional Languages*, 2001.

[6] Peyton Jones, S. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, Cambridge, UK, 2003.

[7] Somogyi, Z., Henderson, F., and Conway, T. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3), 1996, 17-64.

[8] Turner, D. Miranda: a non-strict functional language with polymorphic types. *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, 1985, 201, 1-16.