Slicing wxHaskell modules to derive the User Interface Abstract Model

Daniela da Cruz[1] and Pedro Rangel Henriques[1]

University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{danieladacruz,prh}@di.uminho.pt

Abstract. In this paper, we discuss an experience slicing an Haskell program to separate its GUI developed with the graphic toolkit wxHaskell. The slicing operation was implemented in Strafunski, an Haskell-centered software bundle for generic programming and language processing. To analyze the user interface and extract the abstract model, we first parse the source Haskell program, and build its Abstract Syntax Tree (AST). Then a strategic traversal of the AST is used to slice the GUI wxHaskell code, and build the graph of dependencies between widgets. In this way it becomes possible to understand the execution flow of the program.

Reusing and modifying legacy systems are complex and expensive tasks, because the required program comprehension is a difficult and time-consuming process. Thus, the need for methods and tools that facilitate program comprehension is urgent and strong. Reverse engineering aims at analyzing the software in order to represent it in an abstract format that makes easier to understand *what it does* and *how it works*. Reverse engineering tools provide means to support this task. Nowadays those software systems offer a rich interaction with the user, and their architectures are oriented towards a layered organization, separating at least the interface layer from the business core and data persistence modules.

In this context—system maintenance and comprehension of software with graphical user interfaces—we present the results of a research work on the application of *strategic programming* and *slicing techniques* to the reverse engineering of interactive functional applications. Holmes, the tool so far developed and described in the paper, is capable of deriving the *user interface abstract model* for interactive functional programs. Since wxHaskell became the most famous graphic toolkit to build Graphical User Interfaces (GUI) for Haskell, we have investigated slicing techniques specialized for wxHaskell modules.

In order to extract the user interface model from a Haskell/wxHaskell program we need to isolate the subset of functions responsible for the interaction. *Slicing techniques* are appropriate for that task. To build Holmes it is necessary to construct a slicing function [1, 2] that isolates the wxHaskell subprogram from the entire Haskell program. At a first glance, the obvious and easiest way is to define a recursive function to traverse the Abstract Syntax Tree (AST) of the program and return the wxHaskell subtree (maybe a forest). However, that approach forces the programmer to have full knowledge of Haskell grammar, and to write a complex and long set of mutually recursive functions. *Strategic programming* is based on the use of a set of generic tree-walker functions that visit any AST, enabling the programmer to concentrate just on the relevant nodes; different traversal strategies (e.g. top-down, bottom-up, ...) can also be used. So this approach seems very convenient for our specific purpose.

Classic software systems have a *hierarchical graphical front-end* (the upper layer) that interacts with user reading his input, and writing system messages to him (results or general info). We can say that the interactive layer produces, deterministically, graphical output from *user or system events*, according to the system internal state. A graphical user interface (GUI) is built up from *widgets* (graphical objects), each one with a fixed set of attributes. At any time during the execution of the GUI these attributes have discrete values (the *widget properties*), the set of which compose the *state* of the GUI.

External (user originated) or internal (system originated) events are processed by the other layers and produce changes in the attribute values (alter the widget properties), implying the transition to a new state. So the behavior of the user interface can be described by a sequence of state transitions. A *dependence-graph*—a *directed-graph where nodes are that states, and events label the arcs*—is a perfect *abstract model* to describe GUIs.

To create that GUI abstract model from a given Haskell program we need: to identify *data entities* that denote widgets, and *actions* that denote events; and to discover the *relationships* that interconnect them. For that, we have defined a small set of abstract operations that describe the interactions between the user and the system, i.e., the events responsible for the state changes: *User Selection*–any choice that the user can make between several different options; *User Action*–an action that is performed as the result of user input or selection; *System Output*–any communication from the application to the user. Then we studied wxHaskell library to identify functions and data types that characterize the GUI module. Those identifiers will enable us to slice the user interface component in Haskell programs, isolating the tree nodes that correspond to the GUI. An Haskell GUI is described by types: `Frame` and `Button`, that denote graphical objects, widgets; `text` and `layout`, that represent attributes (functions that applied to widgets return values); `Event`, special attributes that can be transformed into an attribute using the `on` function (the value of an event attribute is normally an IO action that is executed when the event happens); etc.

After the contextualization of the our work and introduction of the basic concepts, we will describe the steps of our approach, and how they were implemented in Holmes.

First Step: Building the AST.

To extract the abstract model for a given source Haskell program, we build the respective AST. It will be our internal representation of the source system that we use in order to isolate the interface layer (wxHaskell sub-program) from the whole application (Haskell program). Any Haskell parser could be used to implement this phase build the AST. As there are many Haskell parsers available, we do not want to develop a new one from scratch. In this project, we use the libraries `Language.Haskell.Parser` and `Language.Haskell.Syntax`. At moment, only a subset (a realistic one) of wxHaskell components is being considered by the Holmes. Our first objective was to explore this approach and to assess its feasibility and usefulness. In the future, we will extend our tool to handle more complex user interfaces. There is no reason to preview any kind of problems in that extension.

Second Step: Slicing Haskell with Strafunski

Given the AST, the next step is to isolate the wxHaskell sub-trees from the whole Haskell tree. As said in the beginning, we decided to follow a *strategic programming* approach to implement this phase, overcoming the disadvantages of a traditional approach to prune the tree.

Strategic programming is a generic programming style for processing compound data such as terms and object structures. Strategic programming was initiated in the setting of term rewriting, but has been transposed to other programming paradigms, most notably functional and object-oriented programming. With strategic programming, one gains full control over the application of basic actions, mainly full traversal control. Using a combinator style, traversal schemes can be defined, and actual traversals are obtained by passing the problem-specific ingredients as parameters to suitable schemes. In this style of programming, there is a set of generic traversal functions that traverse any AST. These strategic functions can traverse into heterogeneous data structures while mixing uniform and type-specific behavior [3]. In this project, Holmes was implemented with Strafunski library [4]: an Haskell library for generic programming and language processing.

As stated above, we have identified the set of types that will help us to recognize the *data entities*, the *actions* and the *relationships* between them necessary to build the GUI abstract model. A deeper study of the grammar, allowed us to know precisely what we need to look for in the list of nodes corresponding to the declarations in the AST. In order to prune the required nodes from whole tree and collect the correct information, we have written in Strafunski a strategy that performs a full traversal (`full_td`, i.e. visits every subterm), in a *top-down* way. To execute a type analysis, the strategy programmed is *type unification* (`applyTU` in Strafunski). The result of pruning the AST with this strategic function is a list of statements of type `HsGenerator`.

Third Step: Building the dependency-graph (the Abstract Model desired).

After slicing the AST, we are able to build the dependency-graph that will model the interaction layer of the given program.

To implement this phase in Holmes, we developed 3 functional strategies: To extract the source node, we explore the information from the first expression in the context of a `HsApp`, until we find a node with type `HsVar`; To extract the action, we look for the name of the widget that appears in the context of a `HsApp` (more precisely, in context of the first `HsVar`); To extract the target node, we consider the node that appears in the context of a `HsQualifier`, and we explore until we find a node of type `HsInfixApp` (but returning the right side of the expression).

This approach has proved to be flexible and broader enough to be applied in real cases of software engineering. The use of functional strategic combinators, make us able to extract not only the wxHaskell sub-program from a Haskell program, but also modules programmed with another GUI toolkits. Actually, we can generalize the functions above referred to other language constructors, by the reason that the basic concepts of strategic programming are independent of the programming paradigm.

Holmes prototype was tested with Haskell programs of small and medium complexity. In all cases, it produced the correct model.

References

1. Tip, F.: A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands (1994)
2. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes **30** (2005) 1–36
3. Lämmel, R., Visser, E., Visser, J.: The Essence of Strategic Programming. 18 p.; Draft; Available at `http://www.cwi.nl/~ralf` (2002)
4. Lämmel, R., Visser, J.: A Strafunski Application Letter. In Dahl, V., Wadler, P., eds.: Proc. of Practical Aspects of Declarative Programming (PADL'03). Volume 2562 of LNCS., Springer-Verlag (2003) 357–375