Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Bachelor Project

ScheMoose
Analyzing and Visualizing Scheme Programs

Katerina Barone-Adesi

supervised by
Prof. Dr. Michele Lanza

Abstract

SCheMoose extracts structural and semantic information from Scheme programs, using a variety of heuristics. We use ScheMoose to analyze and visually reverse engineer 4 Scheme programs, resulting in 4 case studies.

Acknowledgments

I'd like to acknowledge Michele Lanza, Marco D'Ambros, Mircea Lungu, and Romain Robbes for their advice and feedback on this project. Their patient explanations of aspects of Moose and Mondrian, their critiques of weak points of ScheMoose , associated documentation, and visualizations, and their willingness to share their experience in designing visualizations were greatly appreciated.

Contents

Chapter 1

Introduction

A substantial amount of resources are devoted to the analysis of code. According to [Som00], 50-75% of costs in software projects are spent in the maintenance phase. Half of these costs are related to understanding the existing code, aka reverse engineering it. To extract information automatically from programs in the functional language Scheme, we developed a new tool, called ScheMoose .

The reverse engineering of functional programs is a subset of the field of reverse engineering programs; this subset has not been explored to the same degree as other subsets, such as that of the reverse engineering of object oriented and procedural languages. Nevertheless, functional programs are a significant minority of programs. While there are quite a few tools for analyzing functional programs, there don't appear to be any widely used tools at present for reverse engineering them.

Once the decision had been made to analyze functional programs, the first question was which functional programming language to analyze. Some of the more common ones include OCAML[1], Haskell[2], SML[3], various lisp dialects (such as Common Lisp[4], Scheme[5], and Dylan[6]). More unusual contenders include languages like Miranda[7], and various minimalistic languages based on pure lambda calculus.

At Università della Svizzera Italiana, there was already an existing body of code in Scheme, largely due to its use in the first semester "Programming Fundamentals" class. Scheme is fairly widely used, by the standards of functional programming languages, is well-defined, and does not have a baroque syntax. Every other language considered lacks at least one of these criteria; these were the factors which decided which functional programming language to analyze.

1.1 Structure of the Document

Chapter 2 discusses the problem which ScheMoose is designed to address, namely, the analysis of programs written in Scheme.

Chapter 3 describes the solution in detail.

[1] http://caml.inria.fr/
[2] http://www.haskell.org/
[3] http://www.smlnj.org/
[4] http://www.lisp.org/HyperSpec/FrontMatter/index.html
[5] http://www.schemers.org/Documents/Standards/R5RS/
[6] http://www.opendylan.org/
[7] http://miranda.org.uk/

Chapter 4 shows the validation, which is in the form of 4 case studies.

Chapter 5 draws conclusions and proposes a number of areas which are open for future work.

Chapter 2

Reverse Engineering Scheme Programs & Related Work

To reverse engineer a Scheme program, there are several steps. Scheme source needs to be read in, parsed, and analyzed. There also needs to be at least one way to exporting this analysis, whether directly for human use, or into a format for use by other tools.

Reading, parsing, and exporting are fairly straightforward. The interesting part is the heuristics to extract information from Scheme source. There are two problems: what information to gather, and how. We decided to focus on the most important information first, such as function definitions, and call graphs, and to gather this information using heuristics. This is described in more detail in Chapter 3.

Another problem is what to do with the gathered information. A raw analysis is of little help; it is harder to read than source code, and contains less information, as it is derived from it. ScheMoose exports the analyzed information into two formats, for analysis by external tools, namely Moose and dot. To dot, ScheMoose only exports callgraphs. The information exported for analysis with Moose is richer. This is due to the fact that Moose is a reverse engineering environment, and supports formats, such as MSE, which are designed to contain mined informations about programs. The design has a clear separation of concerns, and no formatting information is present in MSE files. Visualizations are created (using Mondrian) from the model Moose creates.

2.0.1 Related work

There is a large body of work on functional programming, and another on software visualization. The most relevant work for ScheMoose is the existing work on Moose, FAMIX, MSE, and Mondrian, as these define the formats and provide the tools to effectively work with the information which ScheMoose mines from Scheme source code.

Moose[1] is a language-independent platform for the analysis of programs. The role of ScheMoose is to export information in the MSE format, so that Moose can leverage Scheme - or, conversely, so that ScheMoose can take advantage of Moose to actually use the information retrieved from Scheme source.

FAMIX is a meta-model. The Software Composition Group [2] wiki entry on FAMIX [3] describes it as "the central meta-model of Moose".

[1]http://smallwiki.unibe.ch/moose/

[2]http://www.iam.unibe.ch/ scg/

[3]http://smallwiki.unibe.ch/moose/famixmetamodel/

MSE [4] is a concrete format, which is used by Moose to import and export data. By default, it uses the FAMIX meta-model. This is the format that ScheMoose exports for analysis by Moose.

Mondrian [5] is a visualization engine. It provides a number of graphical primitives and layouts, and a way to easily script them together to create visualizations of information collected from software, based on a model within Moose. The visualizations which ScheMoose serves as a Scheme backend for are realized using Mondrian scripts.

[4]http://smallwiki.unibe.ch/moose/mseformat/
[5]http://smallwiki.unibe.ch/moose/tools/mondrian/

Chapter 3

ScheMoose

To address the lack of reverse engineering tools for functional programming languages, a first attempt has been made to create one for Scheme, called ScheMoose .

ScheMoose mines information from Scheme source code. This information is both structural and semantic. An example of structural information is the number of global variable definitions, which can be inferred by counting the number of global variables from which information is mined. An example of semantic information is the (heuristically determined) predicate about whether or not a specific Scheme function is a predicate function.

3.1 Architectural Sketch

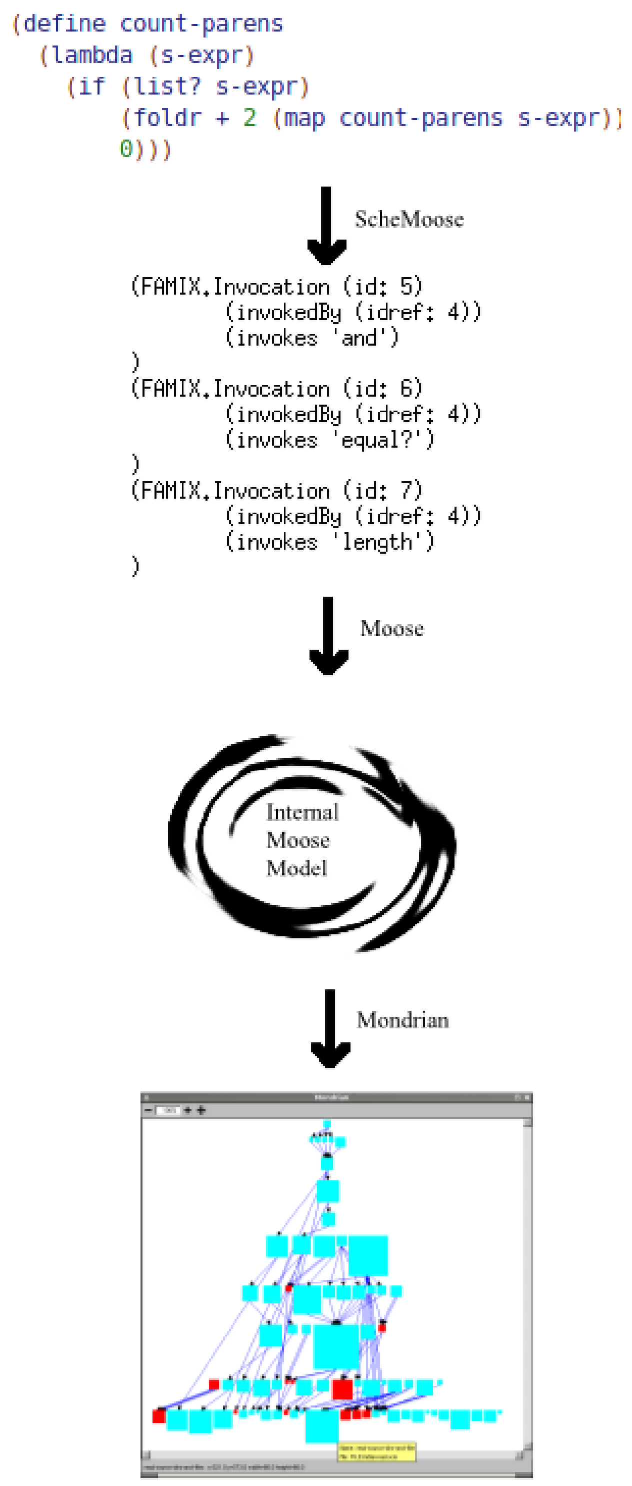
This section provides a high-level overview of ScheMoose's approach, and the role it plays in the process of converting Scheme source code into a visualization. More details are available in the 'Implementation' section, which provides a file-by-file analysis of ScheMoose 's source.

To turn Scheme source code into a visualization, there are 3 steps. The first is to analyze Scheme source code, and export it; this is the main task of ScheMoose . When this exportation is to the MSE format, the next step is to load the MSE file in Moose. This creates a model of it which Mondrian, the visualization engine, can use. This is depicted graphically in Figure 3.1.

Schemoose is a two-pass analyzer, which takes Scheme source code and internally models it information, statement by statement. The first pass builds up the general structure, as well as specific information local to statements, some of which is derived, such as whether a function is a predicate. The second pass tries to resolve calls to their occurances within the model.

Scheme code is read in, and parsed, using the `read` function of DrScheme. This function reads one s-expression at a time from a given file handle, and returns it fully parsed.

Information, in both passes, is gathered by a variety of heuristics. These vary in their reliability, and in how much information they miss. At the low end, information on whether a function is a predicate is based on a common Scheme coding convention (having a name that ends with '?'); this is still sufficient for the majority of programs analyzed. This particular heuristic, for determining predicates, sufficed for 3 of the 4 case studies.



```
(define count-parens
  (lambda (s-expr)
    (if (list? s-expr)
        (foldr + 2 (map count-parens s-expr))
        0)))
```

```
(FAMIX.Invocation (id: 5)
        (invokedBy (idref: 4))
        (invokes 'and')
)
(FAMIX.Invocation (id: 6)
        (invokedBy (idref: 4))
        (invokes 'equal?')
)
(FAMIX.Invocation (id: 7)
        (invokedBy (idref: 4))
        (invokes 'length')
)
```

Figure 3.1: Architectural Sketch, from Scheme to Visualization

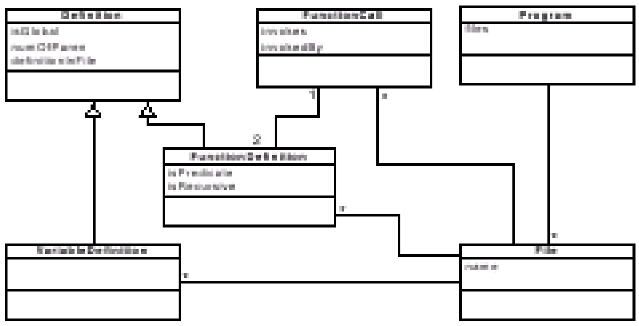


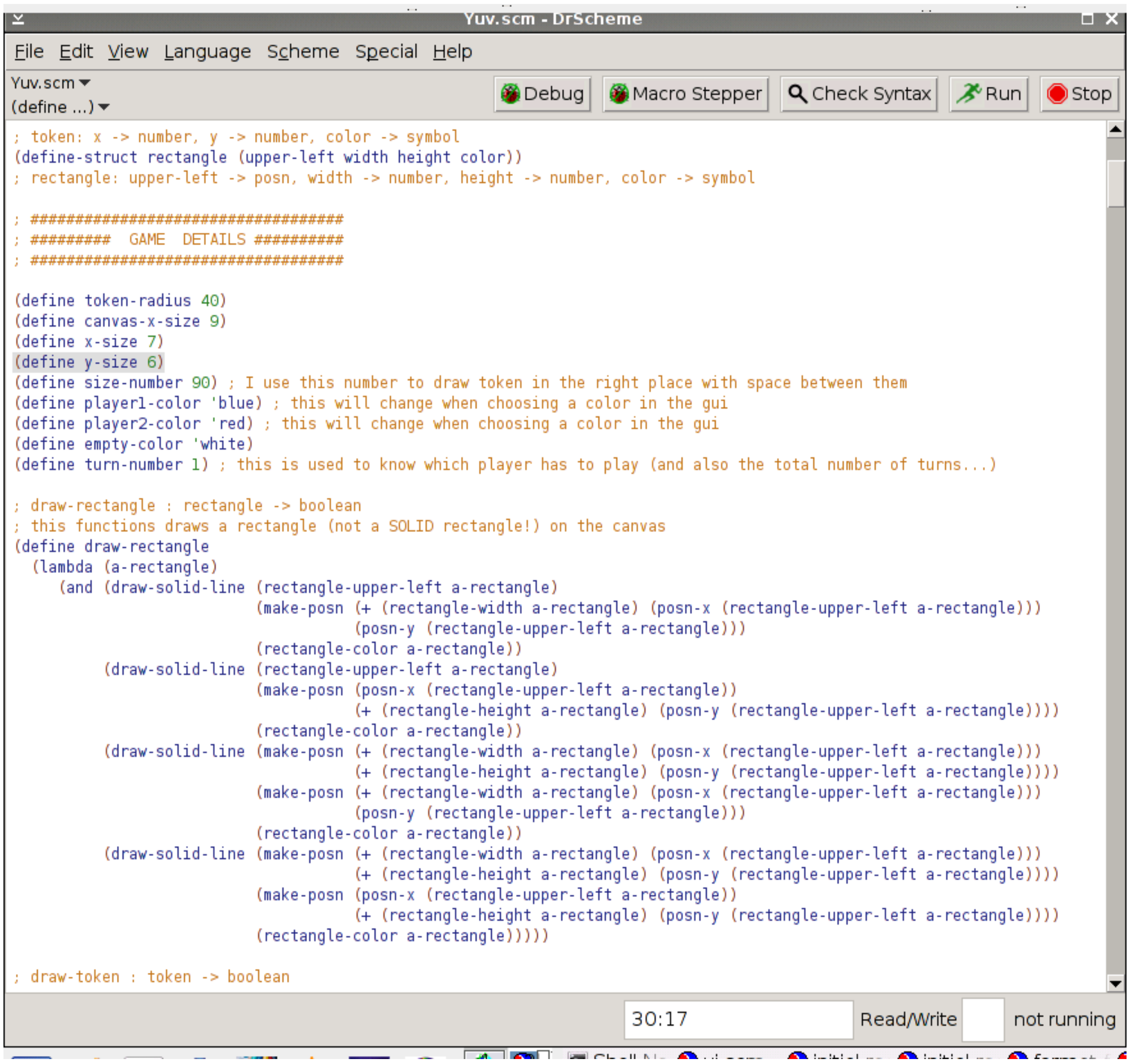
Figure 3.2: The ScheMoose metamodel

Scheme has several subtleties. 'Special forms'/macros allow for the introduction of new syntax. Functions and variables can be defined conditionally and on the fly. Higher-order functions are fairly commonplace. Additionally, Scheme has first-class support for continuations. None of the analyzed projects use them, and ScheMoose has not yet been adapted to model them in any way, but we expect that this would be an intriguing area to investigate.

ScheMoose can export the gathered information in one of two formats. The preferred format is MSE, to allow use of the information with Moose and Mondrian. The secondary format, which currently only exports the callgraph, creates graph descriptions for the program "dot"[1].

Minor changes to Moose were required for this project. These extensions allow it to model the number of parentheses, whether or not functions are directly recursive (that is, a function which calls itself, by name - contrast with mutual recursion, where multiple functions call each other in a circle, or no recursion, where no function is a child of itself in a callgraph), and whether or not functions are predicate functions. The subset of the meta-model used is shown in Figure 3.2.

[1]A component of Graphwiz. http://www.graphviz.org/



```
; token: x -> number, y -> number, color -> symbol
(define-struct rectangle (upper-left width height color))
; rectangle: upper-left -> posn, width -> number, height -> number, color -> symbol

; ####################################
; #########  GAME  DETAILS ##########
; ####################################

(define token-radius 40)
(define canvas-x-size 9)
(define x-size 7)
(define y-size 6)
(define size-number 90) ; I use this number to draw token in the right place with space between them
(define player1-color 'blue) ; this will change when choosing a color in the gui
(define player2-color 'red) ; this will change when choosing a color in the gui
(define empty-color 'white)
(define turn-number 1) ; this is used to know which player has to play (and also the total number of turns...)

; draw-rectangle : rectangle -> boolean
; this functions draws a rectangle (not a SOLID rectangle!) on the canvas
(define draw-rectangle
  (lambda (a-rectangle)
    (and (draw-solid-line (rectangle-upper-left a-rectangle)
                          (make-posn (+ (rectangle-width a-rectangle) (posn-x (rectangle-upper-left a-rectangle)))
                                     (posn-y (rectangle-upper-left a-rectangle)))
                          (rectangle-color a-rectangle))
         (draw-solid-line (rectangle-upper-left a-rectangle)
                          (make-posn (posn-x (rectangle-upper-left a-rectangle))
                                     (+ (rectangle-height a-rectangle) (posn-y (rectangle-upper-left a-rectangle))))
                          (rectangle-color a-rectangle))
         (draw-solid-line (make-posn (+ (rectangle-width a-rectangle) (posn-x (rectangle-upper-left a-rectangle)))
                                     (+ (rectangle-height a-rectangle) (posn-y (rectangle-upper-left a-rectangle))))
                          (make-posn (+ (rectangle-width a-rectangle) (posn-x (rectangle-upper-left a-rectangle)))
                                     (posn-y (rectangle-upper-left a-rectangle)))
                          (rectangle-color a-rectangle))
         (draw-solid-line (make-posn (+ (rectangle-width a-rectangle) (posn-x (rectangle-upper-left a-rectangle)))
                                     (+ (rectangle-height a-rectangle) (posn-y (rectangle-upper-left a-rectangle))))
                          (make-posn (posn-x (rectangle-upper-left a-rectangle))
                                     (+ (rectangle-height a-rectangle) (posn-y (rectangle-upper-left a-rectangle))))
                          (rectangle-color a-rectangle)))))

; draw-token : token -> boolean
```

Figure 3.3: An excerpt of the source of a Scheme program

3.2 Collected Information: Overview

Reverse engineering a system from source code, especially a small system, is possible. However, it is not always an efficient use of time. Compare Figure 3.3 and Figure 3.4. The former displays detailed information, and needs to be pieced together and made sense of; it does not fit on a screen, as shown by the scrollbar. The latter immediately makes the call graph, overall system structure apparent, and suggests functions to look at.

For Figure 3.4 to be useful, it needs to be based on information found in Figure 3.3. This section provides an overview of which data is collected from Scheme projects, and hence can be used in the analysis of the system.

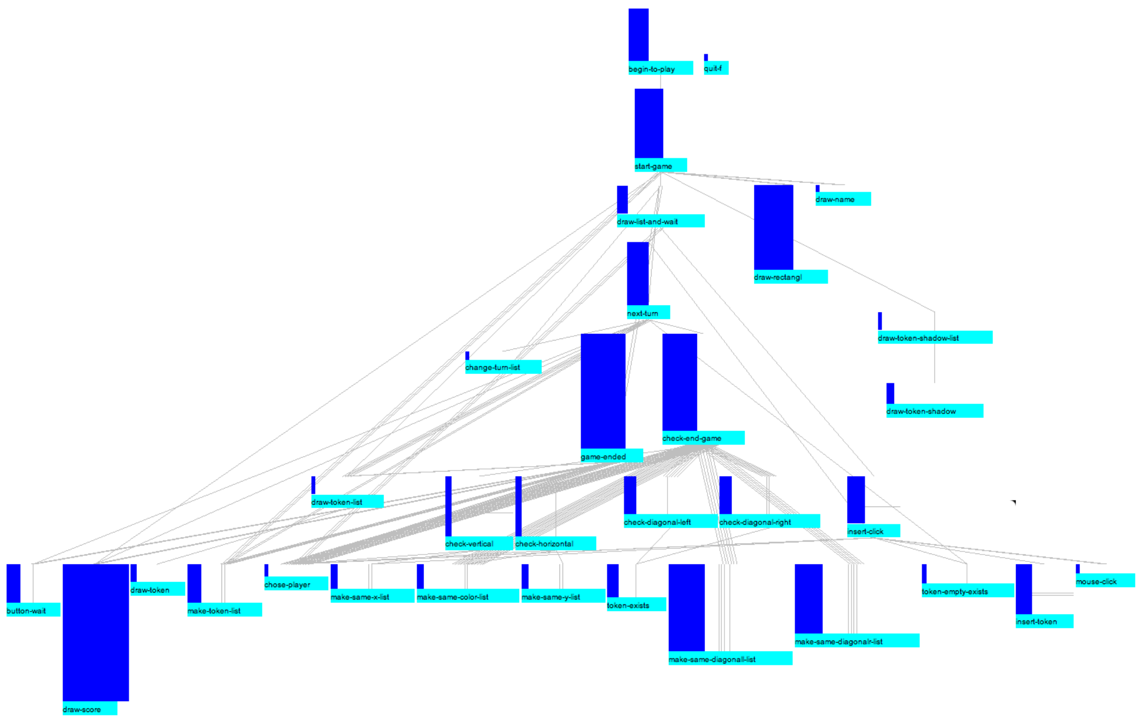


Figure 3.4: Graphical visualization of a Scheme project

For ScheMoose , two categories of mined information were defined: structural information, and semantic information. Structural information is information about the concrete structure of the program. Semantic information is information about the semantics, and is inferred. At present, the information mined includes the following:

Structural:

1. Global function definitions, including the name of the function
2. Global variable definitions, including the name of the variable
3. (Simple) calls between functions, both within the project and to functions defined within the environment.
4. The number of parentheses in definitions.
5. Whether or not a given s-expression is a special form (this is not exported, but it changes how the expression is treated; otherwise, function call information would be wrong).
6. Whether or not a given function is directly recursive.
7. Which file a given definition is in.

Semantic:

1. Whether or not a function is a predicate.

3.3 Collected Information: Detail

This section steps over exactly what information is collected from a Scheme file by ScheMoose . It introduces a small Scheme program, which consists of one global variable and two functions, one of which is a predicate, and another of which is recursive. The predicate indicates if a given number is evenly divisible by the value in the global variable; the recursive function counts how many numbers in a list of numbers fulfill the predicate.

The following subsections explain the information collected in more detail. They describe what is collected, and why, and give an indication of how. Each type of information collected is dealt with individually. The information collected consists of global function definitions, global variable definitions, simple function calls, number of parentheses, special forms, directly recursive functions, files, and predicate functions.

3.3.1 Prerequisite understanding: S-expressions

Scheme syntax is often considered minimal. A Scheme program consists of a number of expressions, each enclosed within parentheses; these are called S-expressions.

S-expressions are essentially a nested list, which corresponds to a parse tree. Scheme code is generally written directly in S-expressions, which makes the mapping simple, as well as rather straightforward to use within Scheme. The Scheme code `(define a 3)` can be treated in Scheme as a list of 3 elements, the first of which is the symbol `define`, the second of which is the symbol `a`, and the last of which is the number `3`.

In a more complex expression, such as `(define b (+ 3 a)`, the same idea applies, but the third element is now the list `(+ 4 a)`, which can recursively be broken into the symbol `+`, the number `4`, and the symbol `a`, in that order. Arbitrary levels of nesting are possible.

3.3.2 Global Function Definitions

Functions are the heart of a Scheme program. They define its structure, both individually (ie, how large a given function is, and other metrics that apply to specific functions), and in the aggregate (ie, the calling relations between them).

ScheMoose collects all top-level function definitions, along with specific information about them (number of parentheses and directly called functions, both of which are described in more detail below). For the purposes of ScheMoose , a top-level definition is any definition of the form `(define asymbol arbitrary-s-expression)`. These definitions are globally visible. Fortunately, in Scheme, it is easy to tell if a definition is a function definition: the arbitrary s-expression has the form `(lambda (list of vars)(function body)`

3.3.3 Global Variable Definitions

Analyzing the global variables of a program is useful, as it gives an immediate idea of the coding style used. Knowing where global variables are declared is useful structural information for figuring out the architecture of a program; this is demonstrated in the lambdaTRON case study in Chapter 4.

ScheMoose collects all top-level variable definitions. These are defined as all top-level definitions which are not function definitions. The information associated with these definitions within ScheMoose is the number of parentheses, and the name of the variable.

3.3.4 Simple Function Calls

Studying functions in isolation provides some information, but a good deal more is found by studying the relation between them. A visual representation of this information can make entry functions, functions which are never called, functions which are called by many other functions, and functions which call many other functions instantly stand out.

ScheMoose collects simple function call information; that is, direct calls, by name, to other functions. These calls are make up a large percentage of the calls in the functions analyzed, and suffice to clearly indicate the general structure of each program. We ignore higher order functions at present; they are significantly more difficult to analyze.

The only tricky part to collecting simple function call information is taking special forms into account. An example of this is analyzing what happens within a `cond` special form. `cond` clauses are in the form `(cond ((predicate) result-if-true)`. A full `cond` has arbitrarily many clauses, and an optional else clause at the end: `(cond clause1 clause2 ... clauseN (else default-result))`. If interpreted by the normal rules, the function call information would be broken: `cond` itself would be considered a function, and `((predicate) result-if-true)` would be interpreted as a call to the function returned by predicate, with result-if-true as an argument. In concrete terms, a legitimate cond clause would be `((> 3 4) 'example)`, which returns the symbol `example` if 3 is greater than 4. This should be considered to only have a call to >.

Without handling of special forms, it would be considered a call to `#f`, the scheme false value, which is the result of evaluating `(> 3 4)`; it would have an argument of `'example`. Incidentally, `#f` is not a function, so this would be an error. The *correct* way to interpret this is to know that `((> 3 4)` actually evaluates `(> 3 4)`, and that the other layer of parentheses are introduced because it is a special form, and delimit the clause. The only way to know this is for ScheMoose to be aware of the syntax of special forms, including `cond`; it is, to some degree, as discussed below and in an appendix.

3.3.5 Number of Parentheses

The Number of Parentheses metric comes from the same idea as the classical Lines of Code metric (which ScheMoose does not implement). It provides a simple way to indicate how 'large' or 'long' a function is. Number of Parentheses has several advantages, especially in Scheme; the largest one is that it is formatting-independent. The number of parentheses does not vary with the spacing used; and, significantly, in Scheme, each parentheses carries semantic information.

In C-like languages, `(an-expression)` is the same as `an-expression`; at most, it prevents an ambiguity in a larger context. In Scheme, these forms are distinct; the former is a function call, and will produce an error at run-time unless the expression is a callable function. Likewise, `((an-expression))` is valid Scheme code, which executes the function returned by executing an-expression; this concept does not have a simple analog in C- or Java-like languages. This explanation intentionally glosses over what happens if the context of the expression is a special form in Scheme; this is explained in the next subsection, 'Special Forms'.

The 'how' of this metric is straightforward: given an s-expression, the number of parentheses within it is found by counting the number of lists it contains, recursively, and multiplying by two. The S-expression `(define a (list 1 (list 2) (list 3)))`, serves as an example. The expression itself is a list, which contains the list three-element list `(list 1 (list 2) (list 3))`, which in turn contains two lists, `(list 2)` and `(list 3)`. This is a total of 4 lists. Accordingly, there are 8 parentheses.

3.3.6 Special Forms

Special forms are the Scheme way of defining custom syntax, and changing the evaluation rules. They are necessary, as Scheme is an applicative-order programming language. This means that arguments to functions are evaluated before the function is called. Without special forms, `(define a 3)` would not work, as `a` would need to be evaluated, but it is not defined. Likewise, `if` would be incorrect, because `(if (a-predicate) action-if-true action-if-false`, as both `action-if-true` and `action-if-false` would be evaluated; if either has side effects or does not terminate, the results would be different from what one expects of an `if`.

All of the Scheme programs analyzed use special forms. All of the first-year programs analyzed used 'local' and 'cond', as did ScheMoose until it was rewritten to use 'let' (another special form) instead of 'local'. Custom special forms can also be defined, but none of the analyzed programs took advantage of this.

ScheMoose has rudimentary support for special forms. The information about special forms is never exported, but it is vital for correctly handling function call analysis in Scheme. This is done by keeping a list of common special forms, which largely follow a few syntactic patterns, organized by pattern. The code for finding function calls works from the outside of an s-expr inwards, and hence knows when it is in one of the known special forms, and handles it according to the syntactic pattern. This approach is limited, but has several advantages: it's simple to implement, to update for new special forms, and to check statically.

3.3.7 Directly Recursive Functions

Scheme discourages iteration in favor of recursion. Consequently, many Scheme programs define quite a few recursive functions. By modelling whether functions are recursive, the style of the program can be better understood.

ScheMoose models direct recursion, which is the most common kind; it is when a function calls itself, directly, by name, within its body. This can be contrasted with mutual recursion (2 or more functions which call each other), and with recursion via higher-order functions; neither of these are modelled.

Finding out whether a function is directly recursive is done by checking if its name occurs in the list of functions it calls.

3.3.8 Files

Scheme programs largely reside in files. Consequently, being able to associate file information with functions is fairly important. Being able to visualize a Scheme program with respect to files gives an idea of the logical layout of the program; visualizing the calls within and between files gives an instant view of the coupling.

SchеMoose largely works in terms of files. The first pass is done on a file-by-file basis, making it easy to associate functions with the files in which they are defined.

3.3.9 Predicate Functions

A predicate function is one which, when evaluated, gives a boolean answer. Predicate functions are used extensively in Scheme; the first argument to the `if` special form, and within each `cond` clause, are predicates. Many predicates, but not all, have names which end with `?`; this is a Scheme convention. SchеMoose considers every function which has a name ending with `?` to be a predicate, and every other function not to be. This heuristic works for 3 of the 4 projects analyzed; the fourth does not follow this convention. A significantly better analysis would require a type inferencing engine for Scheme; the scope of SchеMoose definitely does not extend this far.

Chapter 4

Validation

ScheMoose has been validated by analyzing several programs. The most extensive case study is that on a Connect-4 program written in the first semester by another student, which appears in [BAL07]. Case studies have also been done on 3 other programs. Two are projects from the first semester: “Extreme Ski” and “Lambdatron”. The last is ScheMoose itself.

The case studies share a common thread. In each, a relatively small Scheme program is analyzed, largely with respect to a particular syntactic or semantic factor. Such factors include displaying the calling information of the program, both globally and by-file, showing which files global variables are declared in, and showing which functions are predicates and which are not.

Project name	Source code lines	Number of files	Number of Functions
Connect-4	664	1	32
lambdaTRON	570	7	27
Extreme Ski	975	2	76
ScheMoose	988	15	77

Notes:
There are actually 2 source files in the Connect-4 project, but only one was analyzed, as the second started as a copy of the first. The analyzed source file has 664 lines; the other has 736.
The number of source code lines in ScheMoose has been subject to change during the course of this document.

4.1 Connect-4 Analysis

Figure 4.1 shows the call graph of this Connect-4 implementation. Leaf nodes are functions which only call functions defined outside of the project (or which make no calls, or which only call functions which are not globally visible). These functions are defined by DrScheme, the programming environment which was used. The root node is where the execution of the program begins. This type of visualization is especially useful in languages which do not have a coding convention for the entry point that corresponds to the concept of 'main' in various C-like languages, as it provides instant visual information on where to start tracing calls, and on the relations between functions, which may be arbitrarily arranged in the source.

The callgraph makes a couple of this program evident. One is that the entry point is `begin-to-play`. Another is that large functions are sprinkled throughout the tree. These functions are worth further investigation; they may be evidence of poorly-written code, and/or central to the game.

As Figure 4.2 shows, ScheMoose integrates cleanly with the existing Moose+Mondrian platform. It is straightforward to change the size of the boxes representing functions based on various parameters. Good options are various measures of the size and complexity of the function, such as how many parentheses it contains, or how many other functions it calls. The only change, relative to the previous figure, is in the Mondrian script, and does not involve ScheMoose at all; the MSE file is exactly the same.

This clean integration is due to the modular nature of the platform: ScheMoose is almost purely concerned with analyzing Scheme code and exporting it to the MSE format. Moose then reads MSE files and builds a model. Tools on top of Moose, such as Mondrian, then work with this model. Minimal changes were needed to let Moose support custom metrics, such as the number of parentheses, which is a formatting-insensitive analog to lines of code.

Showing the names of the functions makes the callgraph extremely wide at the bottom. In Mondrian, this is only a small problem, as it is possible to scroll, although it's preferable to be able to see everything at once. In a static document, this is a larger drawback. Consequently, the rest of the visualizations are shown without function names.

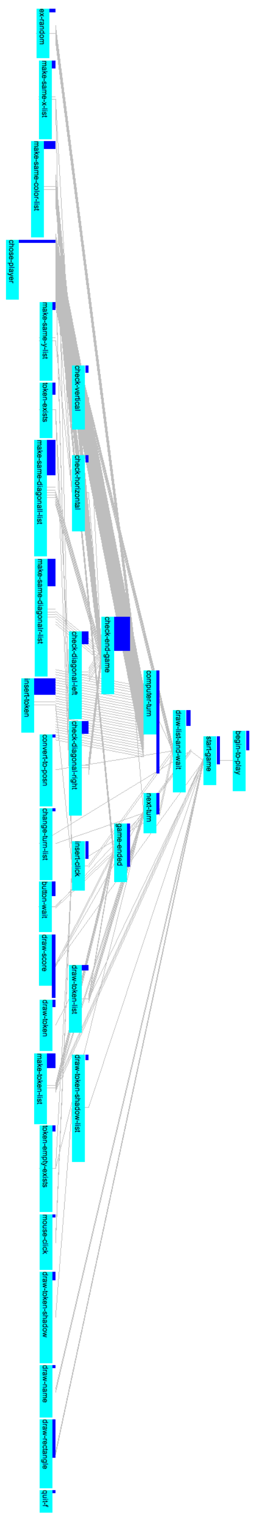


Figure 4.1: The calls of Connect-4

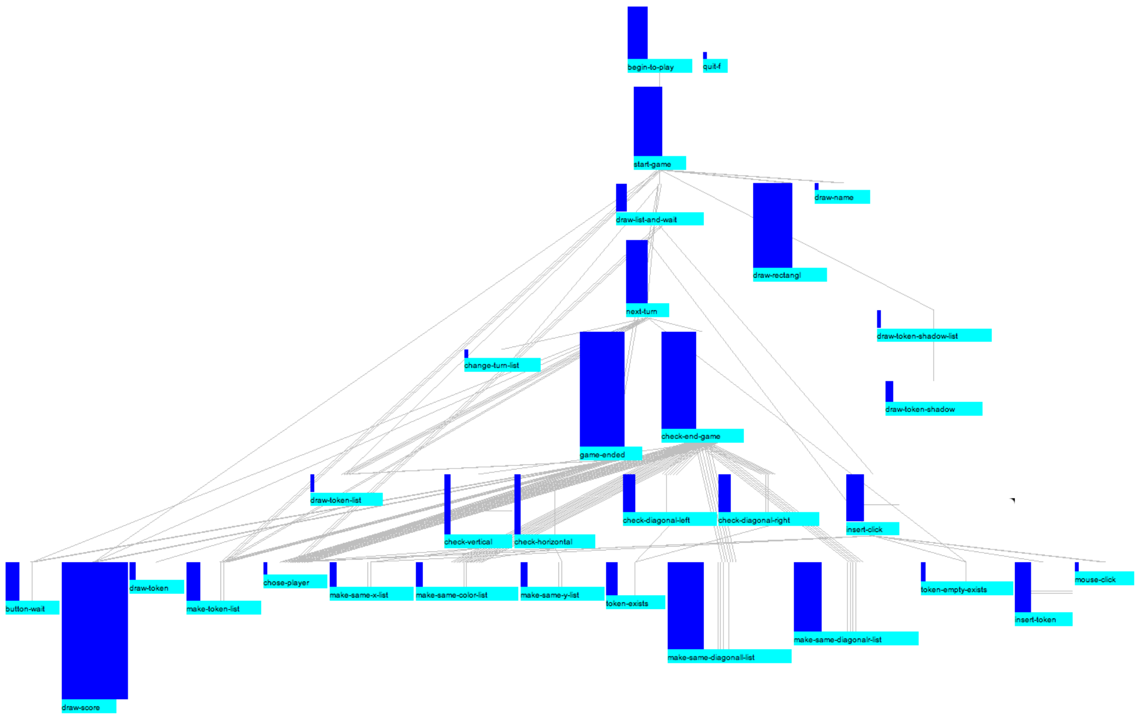


Figure 4.2: The calls of Connect-4, revisited

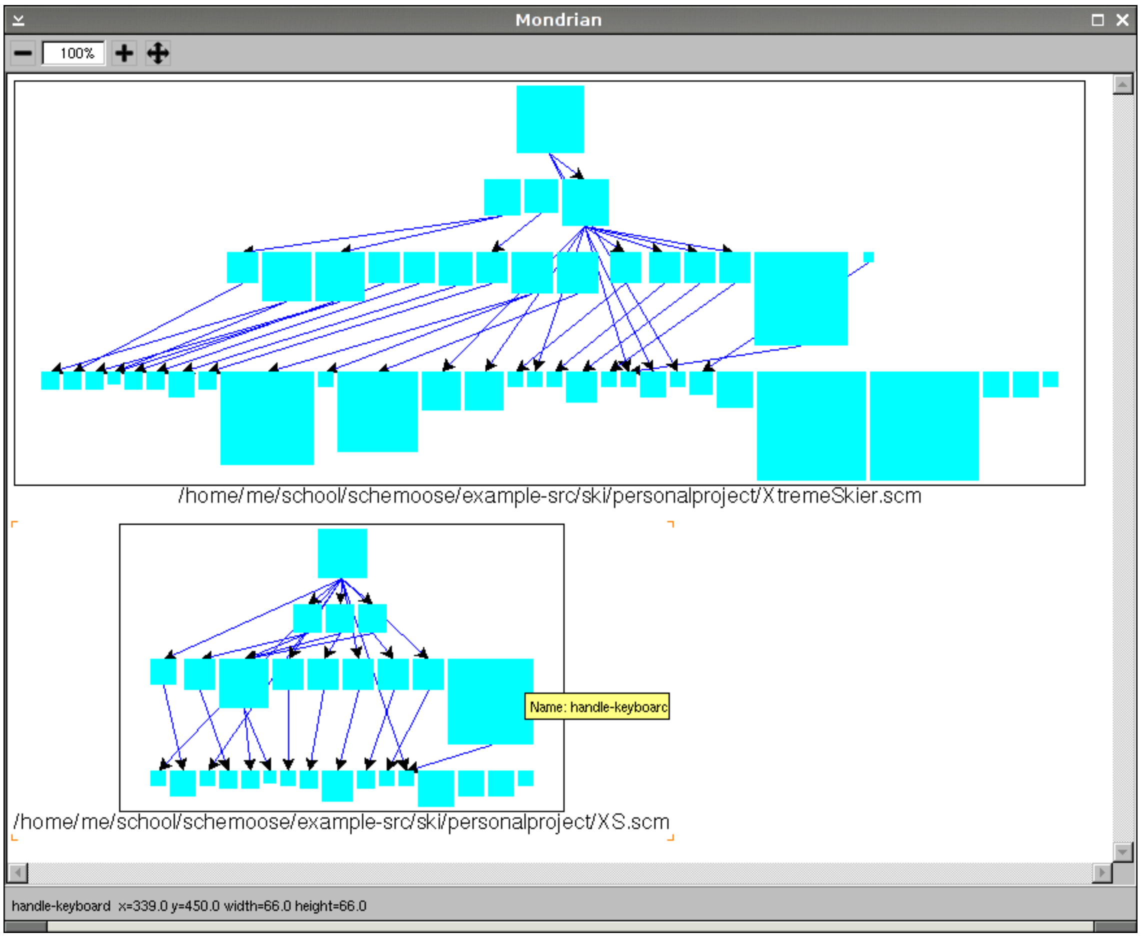
4.2 Extreme Ski Analysis

As Figure 4.3 shows, it is possible to view projects on a file-by-file basis, and to examine the call graphs within each one. This provides a quick clarification as to the relative size, number of functions, size of individual functions, and structure of the call graph of each file.

Figure 4.3 has several properties worth noting. One is the sizes of the squares. Each square represents a function. The width and height are directly proportional to the number of parentheses; this is a rough measure of function size, analagous to lines of code. Larger functions are often worth further investigation, as they are more likely to be complicated, poorly written, or play a crucial role in the program.

Another interesting property is that, in both files, the function calls form a tree. The root of the tree is either the entry point to the program, or a good place to start looking for it.

There are several orphaned functions, in both files. In the particular case of Extreme Ski, this is because these functions are only called from within locally defined functions, and calls from within locally defined functions are not modelled by SchеMoose at present. In other cases, orphaned nodes may represent dead code, which has no callers.


Mondrian
100%
/home/me/school/schemoose/example-src/ski/personalproject/XtremeSkier.scm
Name: handle-keyboarc
/home/me/school/schemoose/example-src/ski/personalproject/XS.scm

handle-keyboard x=339.0 y=450.0 width=66.0 height=66.0

Figure 4.3: A by-file visualization of the callgraphs of Extreme Ski

4.3 lambdaTRON Analysis

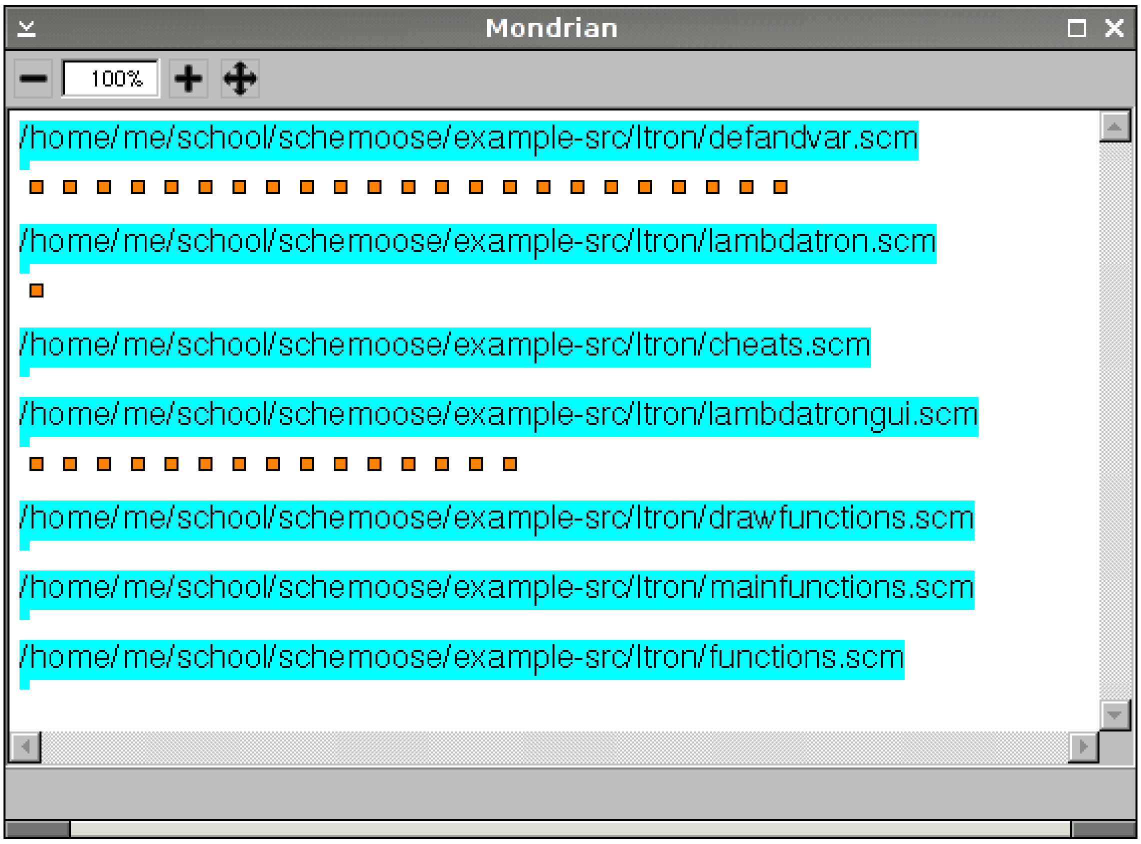
It is widely known, and discussed in the literature [WS73], that global variables have a number of negative effects on a codebase. To understand a codebase, it is useful to know what global variables there are, how they are used, and how they are distributed over the source code.

In this example, a project done by a first semester student, called “lambdaTron”, was analyzed. One area of interest was the use of global variables. As Figure 4.4 shows, they were distributed quite unevenly among the source files; of the 6 source files, 3 had no global variable definitions, one had 1, and the other 2 had over a dozen each. This strongly suggests that the latter two files should be examined.

The file with only one global variable, lambdatron.scm, should also be examined. Immediate questions include “is this global variable necessary?” and “should this global variable be defined in one of the other two files?”. Without the visualization, this outlier would be camouflaged.

Examination of the 2 files which define many global variables shows that they contain what is indicated by their filenames: one is ’defandvar.scm’ and contains definitions and global variables, and the other one is lambdatrongui.scm, and contains the GUI.

One area open for future work in ScheMoose is analyzing the uses of global variables within functions. As Figure 4.4 shows, modelling global variable definitions brings some interesting properties to light, but the information would be greatly enriched by knowing which global variables are actually in use, which files the global variables are used in, what percentage of functions use global variables, etc. This would allow the detection of global variables which are not used (at least directly), ones which are used heavily by many functions, and ones which are only used lightly.



/home/me/school/schemoose/example-src/ltron/defandvar.scm

/home/me/school/schemoose/example-src/ltron/lambdatron.scm

/home/me/school/schemoose/example-src/ltron/cheats.scm

/home/me/school/schemoose/example-src/ltron/lambdatrongui.scm

/home/me/school/schemoose/example-src/ltron/drawfunctions.scm

/home/me/school/schemoose/example-src/ltron/mainfunctions.scm

/home/me/school/schemoose/example-src/ltron/functions.scm

Figure 4.4: The globals of lambdaTRON

4.4 ScheMoose Analysis

Finally, no code analysis project written in the language it is analyzing would be complete without being used on itself. Figure 4.5 clearly shows several elements of ScheMoose , such as the maximum call depth within the project itself, and the relative sizes of different functions. The focus, however, is on whether or not particular functions are predicates; the ones which are are shown in red, while the non-predicate functions are shown in cyan.

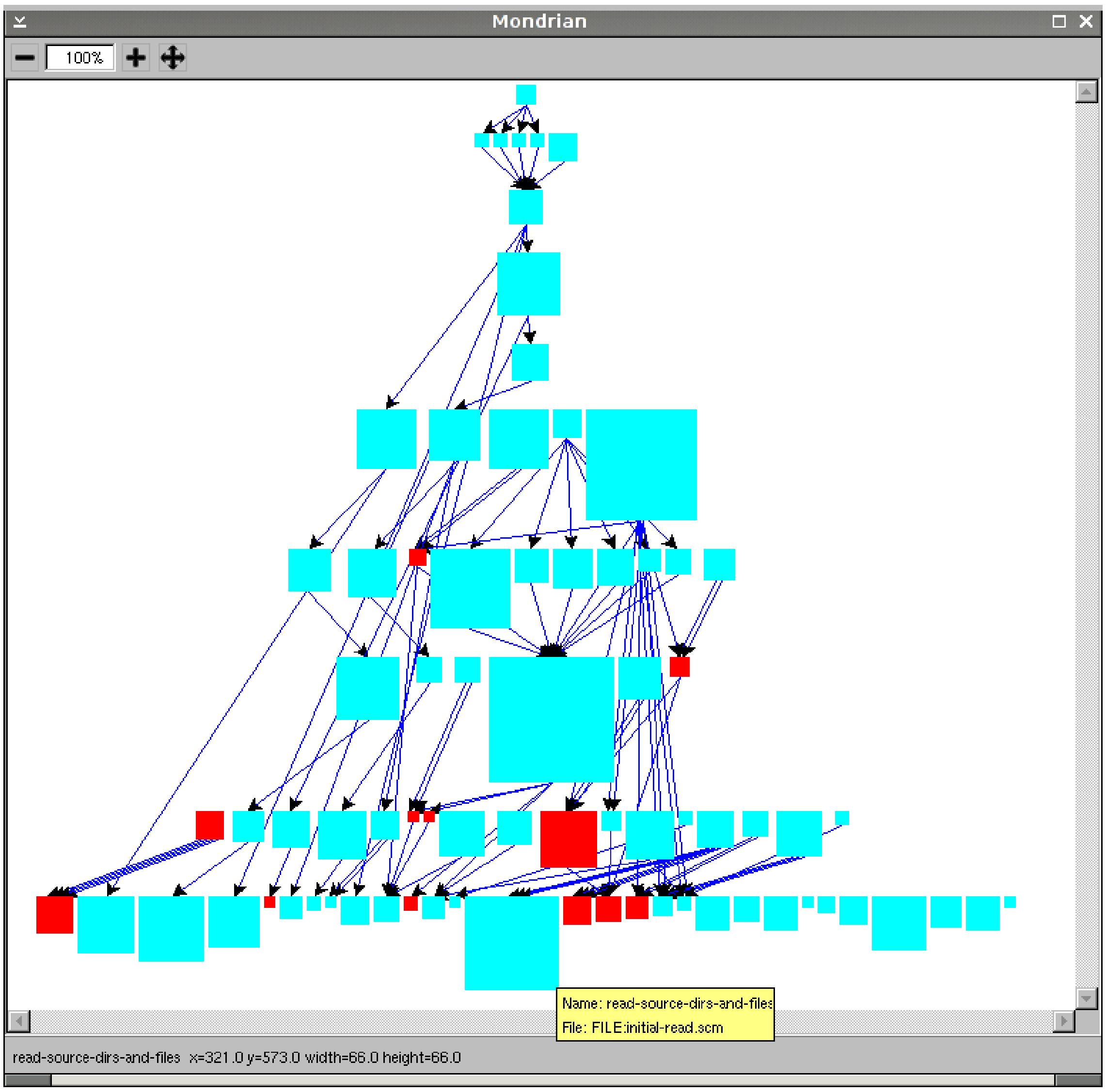
Hovering the mouse over a function shows both its name and the file it is defined in. In this example, the mouse was hovered over the larger red box on the bottom (leaf) row of functions. Figure 4.5 can be contrasted with Figure 4.3: both projects have multiple files, but one visualization shows the callgraphs on a per-file basis, while the other one shows the callgraph of the project as an integrated whole.

Several things within Figure **??** are of note. One is that it appears to have no recursive functions, which is unusual for a Scheme program. The reason for this is that it uses existing (recursive) higher-order functions to perform operations, such as applying a function to every element of a list.

There are several predicate functions within ScheMoose . Notably, they're all either leaf functions, or only call other predicate functions. They are all small functions.

There are several orphaned nodes. Unlike in a previous case study, where orphaned nodes were due to the use of local functions, in ScheMoose , they are largely due to the use of higher order functions. Given a scheme expression such as `(map afunc alist)`, which calls `(afunc anelement)` for each element of `alist` ScheMoose duly records a call to `map` in the MSE file, but does not realize that `afunc` is called once per element of `alist`, and so does not record any calls to `afunc`. This is an example of the use of a higher order function: `map` takes a function as an argument.

The small cluster of nodes at the top is a set of convenience functions, which are now depricated. They serve to re-create mse files for each of the four case studies. Their only differences are the names of the source directories, and of the output files. This symmetry is reflected in their identical sizes. Their caller is a convenience function which regenerates all of them. This information is not immediately available from the visualization, as source code is not presently exported to the MSE file. However, the visualization does suggest these functions as a place to begin investigation. The actual entry point is the slightly larger box to the right of the 4 small convenience functions. This becomes clear when actually using Mondrian, as mousing over it brings up the name of the function, `invoke-main`.



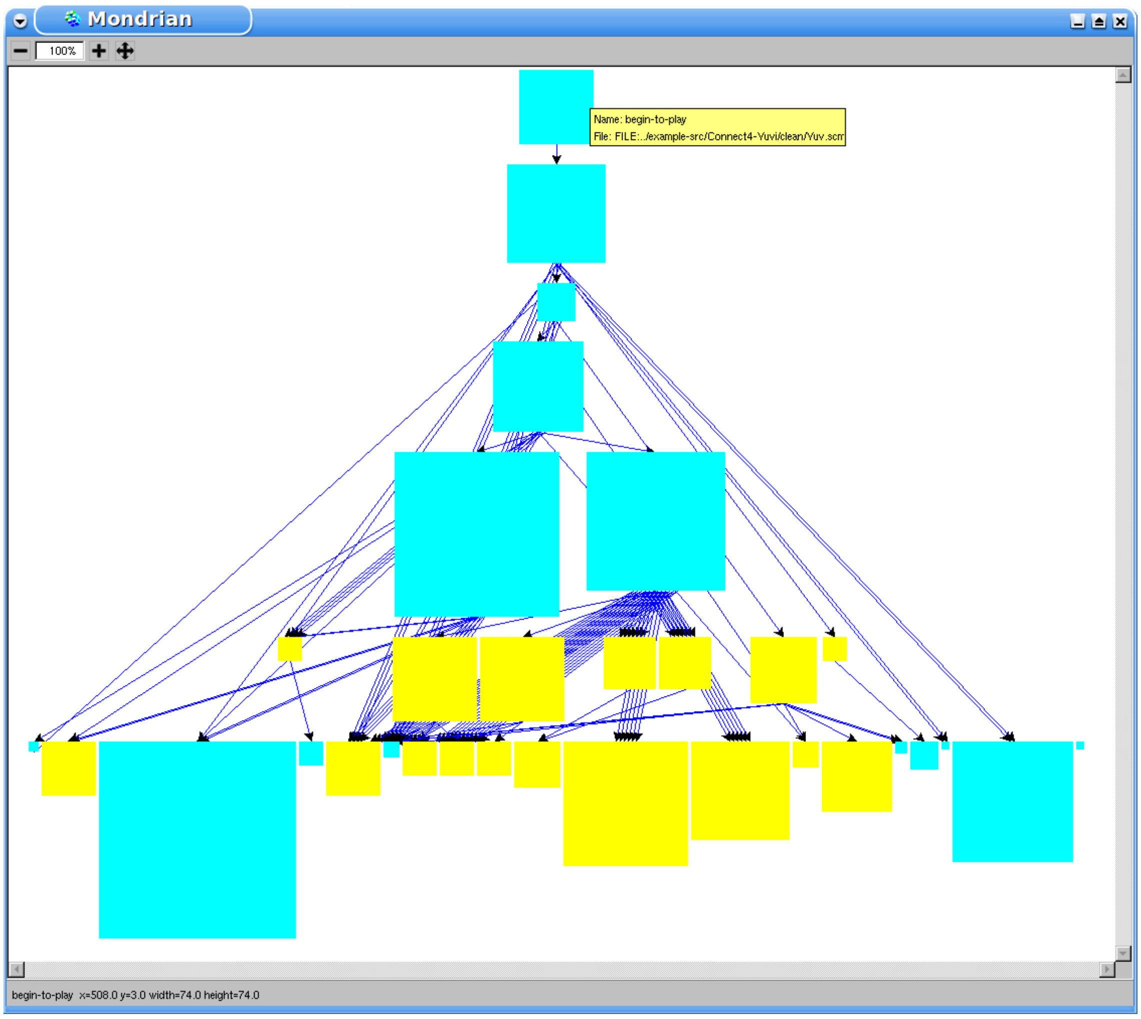
read-source-dirs-and-files x=321.0 y=573.0 width=66.0 height=66.0

Figure 4.5: ScheMoose callgraph with predicates

4.5 Comparison of the 4 programs

In the previous four case studies, specific elements of the programs under examination were discussed. This provided a glimpse of what the information ScheMoose exports can be used for. It is also instructive to look at multiple programs with the same visualization; this illustrates both commonalities and how the programs differ. This comparison is centered on the callgraphs, and the definitions of predicate and recursive functions. The absolute sizes of the functions should not be compared between projects, as the screenshots are scaled to fit this document. Only half of Connect-4 is analyzed, as the two source files it contains are nearly identical, which leads to a rather strange-looking visualizations when they are examined together.

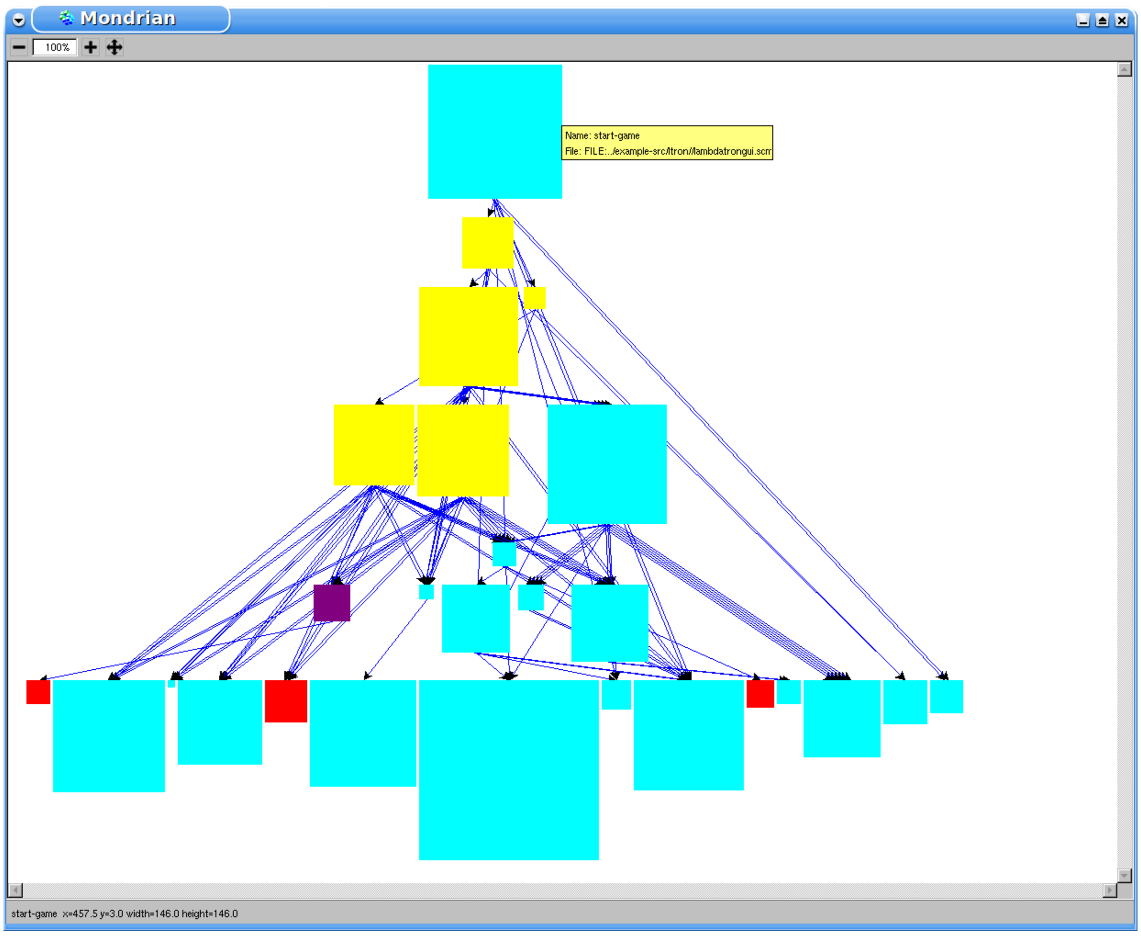
Recursive functions are indicated by yellow, predicate functions by red, and recursive predicate functions by purple. All other functions are cyan.



Name: begin-to-play
File: FILE:../example-src/Connect4-Yuvi/clean/Yuv.scm

begin-to-play x=508.0 y=3.0 width=74.0 height=74.0

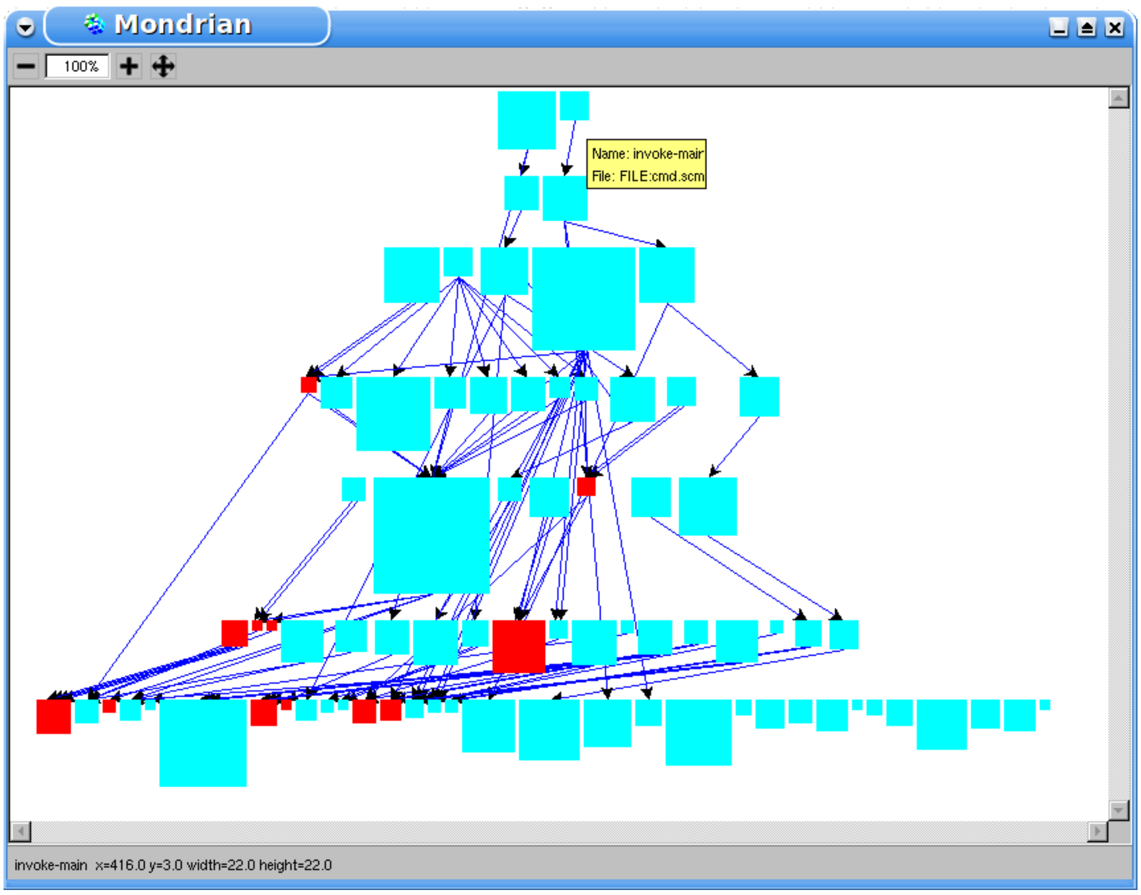
Figure 4.6: Connect-4 (no AI)



Name: start-game
File: FILE:../example-src/ltron//lambdatrongui.scm

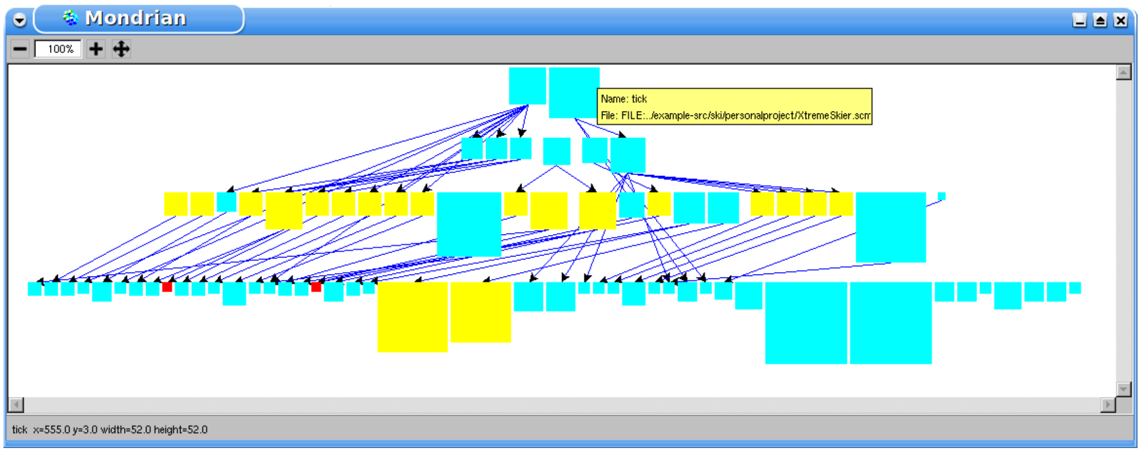
start-game x=457.5 y=3.0 width=146.0 height=146.0

Figure 4.7: lambdaTRON



invoke-main x=416.0 y=3.0 width=22.0 height=22.0

Figure 4.8: SchéMoose



Name: tick
File: FILE:../example-src/ski/personalproject/XtremeSkier.scm

tick x=555.0 y=3.0 width=52.0 height=52.0

Figure 4.9: Extreme Ski

The four programs have significantly different callgraphs. The Connect-4 program is notable for having a few really large functions, and for having functions which call the same second function multiple times (this is represented by multiple lines between the two functions). Extreme Ski is notable for having an extremely shallow callgraph, which has a structure which largely reflects the two files it was written in (compare it to Figure 4.3).

Three of the four projects use direct recursion. The ration of recursive functions are 6/27 for lambdatron, 19/76 for Extreme Ski, and 17/32 Connect-4. This means they make up from just under a quarter to just over a half of the globally defined functions in these projects. The fourth project, ScheMoose , uses higher order functions rather than direct recursion.

Three of the four projects use predicates, marked by the convention of using '?' as the last character of a predicate function name. In all three projects, predicate functions are individually small, and make up a small percentage.

A unique feature of lambdaTRON is that it uses a recursive predicate function. These appear to be rare; even lambdaTRON only contains one.

```
| edges preds recs basics boths|
preds := mooseModel allFunctions select: [:f | f isPredicate and: f isDRecursive not]
recs := mooseModel allFunctions select: [:f | f isDRecursive and: f isPredicate not].
basics := mooseModel allFunctions select: [:f | f isPredicate not and: f isDRecursive
boths := mooseModel allFunctions select: [:f | f isDRecursive and: f isPredicate ].
view interaction popupText:
    [:entity | 'Name: ', entity name, (String with: Character cr),
               'File: ', entity sourceAnchor].
view nodes: preds using:  ((RectangleShape fillColor:  #red asColor)
                        width: [:entity | (entity propertyNamed: #NPAR)]
                                height: [:entity | (entity propertyNamed: #NPAR)]).
view interaction popupText:
    [:entity | 'Name: ', entity name, (String with: Character cr),
               'File: ', entity sourceAnchor].
view nodes: recs using:  ((RectangleShape fillColor:  #yellow asColor)
                        width: [:entity | (entity propertyNamed: #NPAR)]
                                height: [:entity | (entity propertyNamed: #NPAR)]).
view interaction popupText:
    [:entity | 'Name: ', entity name, (String with: Character cr),
               'File: ', entity sourceAnchor].
view nodes: basics using:  ((RectangleShape fillColor:  #cyan asColor)
                        width: [:entity | (entity propertyNamed: #NPAR)]
                                height: [:entity | (entity propertyNamed: #NPAR)]).
view interaction popupText:
    [:entity | 'Name: ', entity name, (String with: Character cr),
               'File: ', entity sourceAnchor].
view nodes: boths using:  ((RectangleShape fillColor:  #purple asColor)
                        width: [:entity | (entity propertyNamed: #NPAR)]
                                height: [:entity | (entity propertyNamed: #NPAR)]).
edges := Core.OrderedCollection new.
mooseModel allFunctions do: [:e | e incomingInvocations do:
[:eInv | edges add: e->(eInv invokedBy)]].
view edges: edges
     from: #value
        to: #key
     using: ((LineShape color: Color blue) decoratedWith: ArrowShape new).
view layout: SugiyamaLayout new.
view open.
```

Figure 4.10: ScheMoose Mondrian code used to visualize all four case studies

Chapter 5

Conclusions

ScheMoose models a significant amount of information about Scheme programs. This information can and has been used to make statements about Scheme programs which would have taken significantly longer to arrive at manually. The visual format used to display the mined information allows for quick comprehension of the structure of a program.

Implementing ScheMoose was largely straightforward, as the simplest design option was usually chosen. This simplicity came at a price: the information mined is not always as complete as it could be, and with even a slightly more complicated implementation, significantly more information could be gathered. The major shortcomings are a lack of modelling of top-level statements which are not definitions, the lack of modelling the internal structure of expressions (especially functions defined within `local` and the various forms of `let`), and the lack of at least simple support (analagous to that for special forms) for higher order functions.

By modelling the top-level statements which aren't definitions, it would be possible to gather the following information, which is currently not gathered: the dependencies between files, and calls to functions from the top level. The dependencies between files can largely be discovered via `load` and `require` statements, which are roughly analagous to `include` or `import` in C or Java. Calls to functions from the top-level are fairly analagous to calls to functions from within other functions, with a few differences, such as being run when the source file is processed by DrScheme, rather than waiting until a specific function of the project is called.

By modelling the internal structure of expressions, it would be possible to have more complete calling information. The first-semester projects, when analyzed, had a few orphan function nodes; manual investigation showed that at least some of these are due to the functions only being called by local functions (that is, functions defined within other functions, which are not in the global namespace). The calls done by local functions are not modelled. The calls to local functions are modelled, but as the local functions themselves are not (only globally defined functions are), they are considered stubs (this is also how calls to functions defined by DrScheme are treated; it means that the function is not modelled by ScheMoose).

Rudimentary support for some of the more common higher-order functions, such as map, filter, and fold, which know that they take an argument which is a function, which they call 0 or more times (once per element of a list which they are also passed), would also allow better calling information. All calls within such structures are currently modelled as calls to map/etc, which they are. However, the calls to the function passed to these higher-order constructs are treated as arguments, and not considered to be called. The analysis of ScheMoose suffered slightly from the lack of this support, as it uses some of these constructs; the first semester projects did not.

5.1 Future Work

ScheMoose only scratches the surface of the analysis of Scheme programs. There are many directions in which it could evolve.

Sticking to Scheme, there is room for the following:

1. Continuations
2. Modelling use of OO extensions to Scheme
3. Modelling use of state
4. Use of I/O
5. Redundant calls
6. Modelling use of higher-order functions
7. Handling of user-defined special forms
8. Modelling mutual recursion
9. Namespace support
10. Modelling the internals of statements.
11. Modelling local variables.
12. Modelling local and global variable use.
13. Modelling top-level statements which are not definitions.
14. Conditional forms of all of the above
15. Better handling of existing special forms
16. Duplicate code analysis
17. Type inferencing (useful for a better predicate heuristic, etc)
18. Heavier analysis techniques, including exhaustive searches and predicate abstraction, to prove various properties hold or do not hold
19. Modelling relevant parts of DrScheme
20. Modelling the relations (load/require) between files
21. Trying ScheMoose with Scheme code designed for interpreters other than DrScheme
22. Examining Scheme code written in unusual styles, such as a continuation passing style, and experimenting with how to best model this.

A few of the above options involve the halting problem. This does not make them useless; it merely means that there has to be a maximum amount to search, and the possibility of an answer which signifies a lack of an answer.

Outside of Scheme, there are quite a lot of functional programming concepts which are not addressed. These include Haskell's concept of monads. While monads can be implemented concisely in Scheme, they are not widely used in Scheme code.

Appendix A

Scheme Source to MSE

This chapter describes how everything modelled by ScheMoose is expressed in the MSE format. This is done via concrete examples, from the case studies. This is done by showing a snippet of Scheme code, followed by the corresponding snippet from an MSE file.

A.1 Context

The MSE format has a simple form. It starts with `(Moose.Model`, followed by `(entity`, and then contains a list of entries for namespaces, files, functions, variables, invocations, etc. Figure A.1 shows a truncated portion of the MSE file generated when analyzing ScheMoose with itself. DIF can be ignored; it's for internal use, to allow an ordering to be seen from within the Moose model. NPAR is the number of parentheses, an approximate measure of size.

A.2 Global Function Definitions

As Figure A.3 shows, the MSE representation of Figure A.2 is initiated with `FAMIX.Function`. Properties are then given, such as an id, a name, the source file it was found in, the namespace it belongs to (as Figure A.1 shows, this is the global namespace) and custom properties, such as NPAR, isPredicate, and isDRecursive.

A.3 Global Variable Definitions

The relation of Figure A.4 to Figure A.5 is much like that of globally defined functions to MSE files. The only real difference is that custom properties that would not make sense (ie, whether or not a function is recursive or a predicate function) are not present.

A.4 Simple Function Calls

A function invocation requires three FAMIX entities in the MSE file - one for each function, and one for the invocation itself. The invocation has an ID, knows what the invoking function is, the name of the function being invoked, and it has at least one

```
(Moose.Model
    (entity
        (FAMIX.Namespace (id: 2)
            (name 'Global')
        )
        (FAMIX.File (id: 3)
            (name '/home/me/school/schemoose/src/s-expr-preds.scm')
        )
        (FAMIX.Function (id: 4)
            (name 'definition?')
            (belongsTo (idref: 2))
            (sourceAnchor
                'FILE:/home/me/school/schemoose/src/s-expr-preds.scm')
            (NPAR 18)
            (DIF 4)
            (isPredicate true)
            (isDRecursive false)
        )
))
```

Figure A.1: MSE in context

```
(define definition?
 (lambda (s-expr)
   (and (equal? (length s-expr) 3)
        (equal? (car s-expr) 'define))))
```

Figure A.2: A Scheme function definition

```
(FAMIX.Function (id: 4)
    (name 'definition?')
    (belongsTo (idref: 2))
    (sourceAnchor 'FILE:/home/me/school/schemoose/src/s-expr-preds.scm')
    (NPAR 18)
    (DIF 4)
    (isPredicate true)
    (isDRecursive false)
)
```

Figure A.3: The MSE representation of a Scheme function

```
(define sample-invocation (make-famix-invocation 5 'invoker UNKNOWN 'invokee))
```

Figure A.4: A Scheme variable definition

```
(FAMIX.GlobalVariable (id: 277)
    (name 'sample-invocation')
    (belongsTo (idref: 2))
    (sourceAnchor 'FILE:/home/me/school/schemoose/src/famix-test.scm')
    (NPAR 8)
    (DIF 277)
)
```

Figure A.5: The MSE representation of a Scheme global variable

```
(define definition?
 (lambda (s-expr)
   (and (equal? (length s-expr) 3)
        (equal? (car s-expr) 'define))))
(define definition-name
  (lambda (s-expr)
    (if (definition? s-expr)
        (second s-expr)
        'NOT-a-define)))
```

Figure A.6: A Scheme function call

```
(FAMIX.Function (id: 4)
    (name 'definition?')
    (belongsTo (idref: 2))
    (sourceAnchor 'FILE:/home/me/school/schemoose/src/s-expr-preds.scm')
    (NPAR 18)
    (DIF 4)
    (isPredicate true)
    (isDRecursive false)
)
(FAMIX.Function (id: 27)
    (name 'definition-name')
    (belongsTo (idref: 2))
    (sourceAnchor 'FILE:/home/me/school/schemoose/src/s-expr-preds.scm')
    (NPAR 14)
    (DIF 27)
    (isPredicate false)
    (isDRecursive false)
)
(FAMIX.Invocation (id: 28)
    (invokedBy (idref: 27))
    (candidate (idref: 4))
    (invokes 'definition?')
)
```

Figure A.7: MSE of Scheme function call

candidate for which function is being actually invoked (as there can be multiple functions defined with one name, though at most one of these would actually be associated with the name at any point during the run-time).

A.5 Number of Parentheses

Number of Parentheses information is collected for global variables and global functions. It is a property of these entities. See Figure A.2 and Figure **??** for an example with a global function.

A.5.1 Special Forms

Special form information is not exported from ScheMoose . It is only used to make sure that function call information is correct, as many special forms introduce syntax. It would be possible to model it in a way similar to the way directly recursive functions are currently modelled, by adding information to the MSE file, and an entry to it to the metamodeland telling it what to look for within the MSE file.

A.5.2 Directly Recursive Functions

Whether or not a function is directly recursive is information collected for globally defined functions. It is a property of the function entity. See Figure A.2 and Figure **??**.

A.5.3 Files

File information is collected for global variables and global functions. It is a property of these entities. See Figure A.2 and Figure **??** for an example with a global function.

A.5.4 Predicate Functions

Whether or not a function is a predicate function is information collected for globally defined functions. It is a property of the function entity. See Figure A.2 and Figure **??**.

Appendix B

Implementation

This chapter discusses some salient features of the implementation.

B.1 Source layout

The source of ScheMoose is divided into several files. These divisions are on conceptual boundaries: for example, one file defines types, another defines a (minimalistic) test framework, and another exports gathered data from the internal format to MSE.

This section presents an alphabetical list of the files, along with their purpose.

B.1.1 all-tests.scm

This is a wrapper, which allows all of the tests of the system to be run at once, by running “mzscheme -i all-tests.scm”.

B.1.2 cmd.scm

This is the command-line interface to ScheMoose . It is defined in more detail in “Invocation: Command Line” in the “How to use ScheMoose ” chapter.

B.1.3 dot.scm

This is the dot exporter for ScheMoose . Given the internal model of Scheme programs of ScheMoose , it calculates the call graph, and exports it as a directed graph.

B.1.4 famix.scm

The core of ScheMoose is famix.scm. Given raw s-expressions (similar to a parse tree) of scheme source, it is what gathers all of the information, storing it in the internal ScheMoose format. It runs both passes of the analysis. A typical action of the first pass is to figure out (with the help of s-expr-preds.scm) what kind of statement a given s-expression is (only definitions are modelled). The second pass is concerned with resolving function calls, so that calls to other functions within the project being modelled are modelled internally and can be exported to the MSE file (the conversion is done by format-famix.scm, and the actual exportation via ui.scm).

B.1.5 famix-test.scm

This defines some basic sanity tests for famix.scm. It can be run standalone, or via the all-tests.scm wrapper.

B.1.6 format-famix.scm

This file is responsible for taking a complete internal description of a modelled project, and converting it into a valid MSE description.

B.1.7 initial-read.scm

This file has two responsibilities. The more minor one is, given a list of files and directories, to figure out (recursively) which of the files and directory contents are Scheme files. The major one is done on a per-Scheme-file basis; it converts the raw file contents into a list of s-expressions, using the 'read' function built into DrScheme.

B.1.8 initial-read-test.scm

Basic sanity tests for initial-read.

B.1.9 mzscheme-compat.scm

The full DrScheme environment is richer than that of mzscheme. This file tries to bridge some of the differences, so that ScheMoose can be used from either. In practice, it's a list of statements importing various libraries.

B.1.10 s-expr-preds.scm

This file is full of utility functions (many, but not all, of which are predicates) for dealing with s-expressions. It is how ScheMoose decides if a given statement is a global variable or a predicate, for instance. It also handles metrics, such as figuring out how many parentheses are in any given s-expr.

B.1.11 s-expr-preds-test.scm

Basic sanity tests for some of the s-expression operations of s-expr-preds.scm.

B.1.12 test-framework.scm

A minimalistic test framework. It provides only two functions, run-test and run-tests. The former takes a function (which should be a test function which returns true or false, although this is not enforced), and if the test fails, indicates the failure. run-tests merely runs all of a list of tests. The major shortcoming of this framework is that it has no way to recover from test errors (ie, use of an undefined variable, unexpected error exceptions, etc). Despite this, it has been very helpful.

B.1.13 types.scm

This file defines the core structures for the internal representation of the gathered information, as well as internal 'undefined' value and a predicate to check if a value in the internal model matches the undefined value.

B.1.14 ui.scm

This provides a variety of convenience functions. The most important is dump-famix-from-to, which, given a list of files and directories, models all Scheme files found, and exports the resulting MSE description to a given output file.

Appendix C

How to use ScheMoose

The following two sections are currently accurate, but subject to change. More detailed and up to date information on the prerequisites and use of ScheMoose may be found on the ScheMoose web page[1].

C.1 Prerequisites

To run ScheMoose, several pieces of software are necessary. These include DrScheme, Visualworks, and Moose. The version of DrScheme is not known to be particularly important, but most of the development was done under version 360. Visualworks needs to be 7.4.x; 7.3 is known to not work.

The trickiest part is the Moose requirement, as an old version from the Bern store, rather than the Cincom store, is the baseline, and there are also several minor custom patches which have not been packaged yet. Until this is done, the author needs to be contacted by anyone who would like to try working with ScheMoose's output.

C.2 Invocation: Command line

Assuming that DrScheme is installed correctly and mzscheme (part of DrScheme) is in your path, all that is needed to run ScheMoose over the files and directories of your choice is the following command: mzscheme -i cmd.scm mse inputs out.mse where inputs stand for one or more files or directories, in any order. Directories are searched recursively for .scm files, case-insensitively. The invocation to export the callgraph as a dot file is identical, except for the format specifier; instead of 'mse' following cmd.scm, 'dot' must be specified.

[1]http://atelier.inf.unisi.ch/ baroneak/schemoose/

List of Figures

Bibliography

[BAL07] Katerina Barone-Adesi and Michele Lanza. Schemoose - supporting a functional language in moose. In *FAMOOSr (1st Workshop on FAMIX and MOOSE in Reengineering)*, 2007.

[Som00] Ian Sommerville. *Software engineering*. International computer science series. Addison–Wesley, Wokingham [u.a.], 6th edition, 2000.

[WS73] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, 1973.